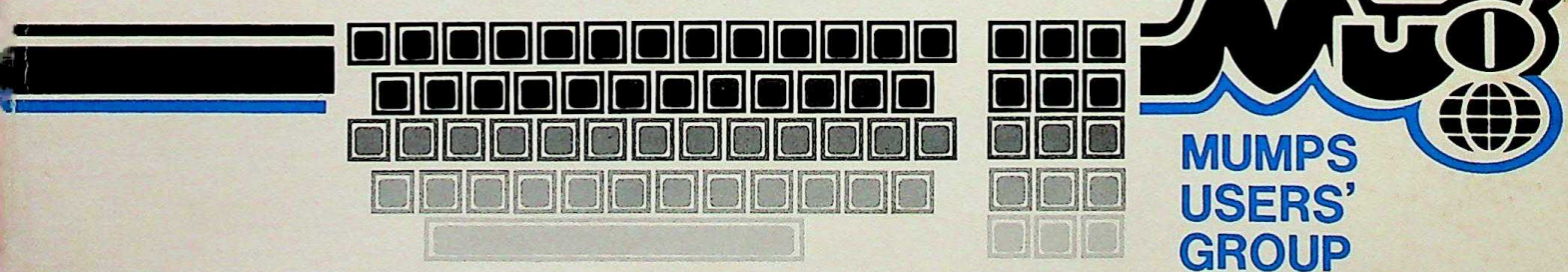
A GUIDE FOR
THE NOVICE

INTRODUCTION TO

STANDARD MUMPS 1984



MUMPS
USERS'
GROUP

Organizations Providing Generous Support for the Publication of this Book:

Digital Equipment Corporation
Two Iron Way
Marlboro, MA 01752
(617) 467-2351

The Medical Systems Group of Digital Equipment Corporation provides applications and systems to the world-wide health care market. The Group coordinates development, sales, and support activities for all of Digital's products for the industry. Digital has played an active role in the development of MUMPS systems since 1969. Digital offers 2 MUMPS products: DSM-11, a multi-user operating system and MUMPS language interpreter for the PDP-11 family of processors, VAX-11 DSM, a MUMPS language implementation layered on VAX/VMS computer systems.

InterSystems Corporation
210 Commercial, 3rd Floor
Boston, MA 02109
(617) 227-1555

InterSystems Corporation develops and markets ANSI Standard MUMPS products for the DEC computer family. The M products satisfy a wide range of needs, from single user systems running under a general operating system (M/PRO for the PRO 350), to high performance compiled MUMPS in a dedicated PDP-11 environment (M/11 +), to systems requiring a large number of terminals in a multilingual environment (M/VX for the VAX). Additionally, InterSystems provides increased flexibility with networking software that allows the creation and use of distributed MUMPS databases.

Micronetics Design Corporation
932 Hungerford Drive, #11
Rockville, MD 20850
(301) 424-4870

Micronetics Design Corporation offers multi-user MUMPS operating systems for microprocessors including Motorola 68000 and 6809, Intel 8086 and 80186, as well as several other microprocessor types. Hardware configurations range from small floppy disk systems thru large Winchester disk systems with tape drives. Implementations are available for Motorola Information Systems (Four-Phase), Pixel, Plexus, Perkin-Elmer, Apple, IBM PC, Radio Shack, and many more including most UNIX* based systems. Available applications include word processing, financial and general business packages, and medical applications including COSTAR.

Plus Five Computer Services
765 Westwood Drive
Clayton, MO 63105
(314) 725-9492

Plus Five Computer Services offers UNIX MUMPS, a MUMPS implementation that runs under UNIX* Version 7 or later or an equivalent operating system; this implementation is available on a very wide range of hardware.

PURVIS Systems Incorporated
6901 Jerico Turnpike
Syosset, NY 11791
(516) 364-9690

PURVIS Systems, specialists in industrial laboratory information management systems, provides custom programming for interactive on-line applications, performs contract work and offers a full range of consulting services.

*UNIX is a trademark of Bell Laboratories

A GUIDE FOR THE NOVICE

INTRODUCTION TO

STANDARD MUMPS

Revised 1984

Revision by Thomas C. Salander
Original Version by Joan Zimmerman

Copyright 1984 MUMPS Users' Group
4321 Hartwick Road, Suite 510
College Park, Maryland 20740

ISBN 0-918118-27-1

TABLE OF CONTENTS

No.		Page
1.	FIRST STEPS	1
1.0.	Introduction	1
1.1.	Signing-On	1
1.2.	The WRITE Command	2
1.3.	Whoops!	6
1.4.	Variables	7
1.5.	READ	8
1.6.	SET	9
1.7.	KILL	10
1.8.	Avoiding An Argument	11
1.9.	Exercises	11
2.	WRITING A FIRST ROUTINE	13
2.0.	Introduction	13
2.1.	MUMPS Example: A Method Of Calculating The Individual Benefits of Exercise	13
2.2.	Introduction To The IF Command	16
2.3.	Arithmetic Operators: The Common Ones	17
2.4.	Truth Values And The Negation Operator	18
2.5.	How to Defer Execution of Your Code	19
2.6.	The GOTO Command For the Permanent Transfer of Execution	21
2.7.	The QUIT Command	23
2.8.	The DO Command For the Temporary Transfer of Execution	23
2.9.	MUMPS Example: Solution	25
3.	MORE ON HOW TO PROGRAM	29
3.0.	Introduction	29
3.1.	Storage of MUMPS Routines	29
3.2.	Retrieval of Code	30
3.3.	Printing A Routine	31
3.4.	Deletion Of Lines Of Code	32
3.5.	Insertion Of Lines	34
3.6.	BREAK Command And Code Editing	34
3.7.	Division of Code	37
3.8.	HALT Command	37
4.	FUNCTIONS	39
4.0.	Introduction	39
4.1.	Expressions	39
4.2.	$LENGTH	40
4.3.	$FIND	41
4.4.	$EXTRACT	44

TABLE OF CONTENTS
(continued)

No. Page

4.5. $PIECE....45
4.6. $LENGTH - Piece Counting....49
4.7. Summary....50

5. DATA....53

5.0. Introduction....53
5.1. Local And Global Variables....53
5.2. Subscripts....54
5.3. Concatenation....58
5.4. $DATA....59
5.5. $ORDER....62

6. MORE COMMANDS....69

6.0. Introduction....69
6.1. Review Of Commands For Manipulating Data....69
6.2. Review Of Four More Commands....70
6.3. The ELSE Command....71
6.4. The FOR Command....73
6.5. The HANG Command....74
6.6. The Timeout....75

7. STYLE IN USER INTERACTION....77

7.0. Introduction....77
7.1. Prompts And Explanations....77
7.2. New Line, New Page, and Tabbing....79
7.3. $JUSTIFY....82

8. ANSWER CHECKING....85

8.0. Introduction....85
8.1. The "Contains" Operator....85
8.2. The "Pattern-Match" Operator....86
8.3. Logical Operators & (And) And ! (Or)....91

9. ASCII CODE: MORE FUNCTIONS AND OPERATORS....95

9.0. Introduction....95
9.1. The ASCII Coding Scheme....95
9.2. $ASCII....98
9.3. $CHAR....99
9.4. Follows Operator....100
9.5. $ORDER and String Subscripts....102
9.6. The ASCII Code and Pattern Matching....103
9.7. Arithmetic Operators....104

TABLE OF CONTENTS
(continued)

No. Page

10. SPECIAL VARIABLES AND MORE FUNCTIONS....107

10.0. Introduction....107
10.1. What is a Special Variable?....107
10.2. $HOROLOG....108
10.3. $IO....110
10.4. $JOB....110
10.5. $STORAGE....111
10.6. $TEST....111
10.7. $X And $Y....112
10.8. $RANDOM....114
10.9. $SELECT....114
10.10. $TEXT....116

11. WRAP-UP....119

11.1. Introduction....119
11.1. OPEN, USE, And CLOSE....119
11.2. LOCK....121
11.3. READ - Fixed Length....123
11.4 READ - Single Character....123
11.5 VIEW and $VIEW....124
11.6. Non-Standard Commands, Functions, And Special Variables..124
11.7. Post-Condition....125
11.8. Abbreviations....128
11.9. Portability Requirements....129

12. THE XECUTE COMMAND, THE INDIRECTION OPERATOR, AND THE JOB COMMAND....131

12.0. Introduction....131
12.1. XECUTE....131
12.2. Indirection Operator....133
12.3. JOB....136

Appendix 1. MUMPS Programming Manual....139
Appendix 2. Implementation-Specific Features....141
Appendix 3. Solutions to Problems....145
Appendix 4. Alternate Code....157

Index....159

CHAPTER 1

FIRST STEPS

1.0 INTRODUCTION

It is essential to begin an introduction to MUMPS by reviewing how to start communicating with a MUMPS computer. The process of making initial contact is called "signing on". An example is described in the first part of Section 1.1, with space provided for you to write in this book the facts that describe your own "sign-on procedure".

Section 1.2 introduces one of the most frequently used commands, the WRITE command. If a terminal is available, you should sign on and try this command immediately: try out all programming features as you learn them. Methods for correcting errors made in writing MUMPS are in Section 1.3.

Variables are an essential component of a computer language and are explained in Section 1.4. Three more commands are given: READ (Section 1.5), SET (Section 1.6), and KILL (Section 1.7). Each command usually acts upon an "argument" that follows it, but sometimes no argument is necessary as discussed in Section 1.8. The chapter concludes with exercises in Section 1.9.

1.1 SIGNING-ON

The first step in learning to program in MUMPS is to locate the device by which you interact with the computer. That device is called a "terminal". You probably give your instructions to the computer by means of a "keyboard" that forms the front part of the terminal. The keyboard resembles one on a typewriter and the keys respond to a gentle touch. You see displayed each message as you send it to the computer. If you are using a "video" terminal, the message appears on a television-like screen. Alternatively, if you are using a "printing" terminal the message is printed on paper. Responses from the computer to your instructions appear on the same screen or on the same paper.

Your second step is to get information about how to set up the terminal so that it can communicate with the MUMPS computer you want to use. Some examples of how to do this are given in Appendix 2. Use the space provided there to write instructions that apply to your terminal if you wish.

Thirdly you must tell the computer, by means of the terminal, that you want to start programming. Do this in three steps. (Specific example appears later). First signal the computer to

start communicating with your terminal. Then identify for the computer a "user area" in which you want to work. (You probably have been given access to an area on the computer that is reserved for beginning programmers. This allows you to experiment without any danger that you could hurt an important application available in a different user area.) Finally, if the computer asks you to do so, enter a password that proves you are authorized for this user area. (The names of the user area and the password are given by a supervisor or instructor, or by a manager of the MUMPS system.) A sample transaction is as follows:

1. Tell the computer that you want to start programming: press the key marked BREAK. The computer responds by printing "HELLO, I AM YOUR STANDARD MUMPS SYSTEM. THE TIME IS 9:31 A.M. WHAT USER AREA DO YOU WANT?

2. Enter the name of your user area, which may be called, for example TRY. After that you must press the "entry key". This key is usually labelled RETURN and located at the right-hand side of the keyboard. The computer acknowledges that by saying 'OK, PROVE YOUR AUTHORITY. GIVE ME YOUR PASSWORD:".

3. Enter your password, which may be, for example, "HUSH". Again, you must press the entry key afterwards, so the computer knows you have finished. The computer replies with its own symbol (which may be, say, and exclamation mark, "!") to show your password was correct and you may now write MUMPS code.

You may record in Table 1 (or in Appendix 2) the information about how you sign on. If your sign-on procedure is more complex than that described, make appropriate notes.

1.2 THE WRITE COMMAND

You instruct a MUMPS computer to do what you want by giving it "commands". Whatever is to be manipulated by the command is called the "argument" of the command and appears after it. Commands and their arguments form MUMPS "code".

The first command is WRITE. For example, suppose you type on your terminal's keyboard:

```
WRITE "MUMPS IS A GREAT LANGUAGE"
```

and then press the entry key. (From here on, I assume that you press the entry key after you have typed your instructions. If you ever find that the computer is not responding, you have probably forgotten to press the entry key.) The computer responds by displaying:

MUMPS IS A GREAT LANGUAGE

on your terminal. Note that the string of characters to be displayed was entered in quotation marks after the WRITE command. Whatever characters follow a WRITE command and are contained between a pair of quotation marks will be written out, just as they are typed.

Now you try. Type on your keyboard:

```
WRITE "HELLO"
```

What happened? If the computer replies:

```
HELLO
```

you have just written your first successful MUMPS instruction. If nothing happens, try pressing the entry key. (If nothing still happens, you may not be signed on to the MUMPS computer, so go back to the previous section and try to sign on again.)

1a. What key do you press to tell the computer that you want to start programming?

1b. What should the computer reply with?

2a. What is the name of your area?

2b. How is the entry key marked on your terminal. (Remember to press it each time you have finished.)

2c. How does the computer acknowledge your specification of the user-area?

3a. What is your password? (NOTE: you probably should not write down the password, but you may. In any case, please remember it.)

3b. What symbol does the computer reply with, to show you that you may now write MUMPS code?

Table 1. How to sign-on to program in MUMPS.

Alternatively, something different may have happened. Any message printed out that is different from the characters you put in quotation marks may be an "error" message in which the computer indicates that it did not understand you. So try again. Be careful that you spell WRITE correctly: a computer is very simple-minded, so if you misspell WRITE it will not know what you mean. After you have spelled WRITE correctly, be sure to specify one (and only one) space by pressing the space bar. Then put quotation marks, followed by the text that you want, and close the quotation marks. This time when you press the entry key all should go well, and the text specified between the quotation marks should now appear.

Try different characters between the quotation marks. You may use numbers and any punctuation symbols (except for a single quote) as well as letters.

What happens when you enter:

```
WRITE "1+4"
```

The computer responds with:

```
1+4
```

If you remove the quotation marks, and enter:

```
WRITE 1+4
```

something else happens. Try it. You should find that the computer replies:

```
5
```

Thus you can see that the computer can do more than display the characters in a string. It can also do sums. Try other sums, such as:

```
WRITE 49+57
```

Remember to put one space after the WRITE command, but do not put more than one. Just put numbers and the addition sign. Keep trying similar examples, to get the feel for writing MUMPS code. Note also that you may put more than one argument such as:

```
WRITE "2+3=",2+3
```

1.3 WHOOPS!

When writing MUMPS code you may occasionally make a mistake. There are two ways of correcting errors. Either you may delete at a stroke all the characters you typed since you last pressed the entry key, or you may delete singly the last one or more characters that you typed. When you are working on a video terminal, your position on the screen is shown by a small (perhaps blinking) line or rectangle called a "cursor". On a printing terminal, the cursor position is the place where a character would appear if printed now. The cursor moves to reflect not only what is printed but also how errors are being corrected.

On many MUMPS systems, the way to delete all of the characters just typed is as follows. First hold down the "control" key, which is often labelled CTRL and appears at the left of the keyboard on most terminals. Keep holding down the control key and press the key for the letter U. Your terminal responds in some way, probably by moving the cursor to the start of a new line and waiting for you to enter the new desired characters. So that you have a record here of what to do and how the terminal responds, write down how to erase all the characters of a line you have just typed:

and write down how the terminal responds:

When you need to delete just a few of the most recent characters, what you do and the response you see differs among terminals. Typically, there is a single key on the keyboard that you press. It is often marked RUB OUT or DEL for delete. Each time you press it, MUMPS responds by printing a backslash ("\") character or by moving the cursor one character backwards and erasing the last character on the screen; concurrently MUMPS forgets the character thus deleted. Do remember that one space (or blank) counts as one character and must be deleted in the same way as a printed character. Again, write here for yourself what you do to erase the most recent undeleted character that you typed:

and write here how the computer responds:

1.4 VARIABLES

So far you have been looking at "constants". These may be strings of characters shown surrounded by quotes or they may be numbers. For example:

"MUMPS IS A GREAT LANGUAGE"

is a constant or unchanging string of characters. And the number 1 is a constant number.

A most useful tool in MUMPS, as in other computer languages, is something whose value can change as necessary. Such a tool is called a "variable". A variable has a "name" by which it is referenced, and it has a "value" that is assigned to the variable and may be changed at any time. Suppose there is a variable called:

NAME

Its value may be determined by entering:

WRITE NAME

The computer displays:

JOAN

thus showing that the value of the variable called NAME is "JOAN". For someone else, the value of NAME might be "GEORGE" or "HENRY". Note that the quotation marks around "JOAN" are not printed when the value of NAME is written. The quotes are there to help us differentiate between a value like "JOAN" and a variable, which could be named JOAN.

Suppose a variable exists called:

TREE

How do we find its value? We enter:

WRITE TREE

If the result is:

OAK

what is the value of TREE? The correct answer is that TREE has the value "OAK". MUMPS places some limits on variable names that you should know about. Names should be eight characters or

shorter. Any alphabetic character is allowed at any position. The initial character may be the "%" symbol instead of an alphabetic. Any character after the first one may be a numeric digit instead of an alphabetic. Some examples of valid variable names are LOUIS, LOUIS2, %LOUIS, and %1. An An invalid name is 20% because (1) it begins with a digit and (2) it has the "%" symbol at the third position.

1.5 READ

You have probably been wondering how you change the value of a variable. One way is by means of the READ command, which tells the computer that it should ask the user for a value to be assigned to a variable. For example, suppose you enter:

```
READ X
```

(pressing the entry key after you have written this line). The terminal waits patiently for you to enter more characters and press the entry key again. Whatever characters you wrote are assigned to the variable whose name is X as its new value. For example, if you type:

```
READ X
```

and then enter:

```
A PROBLEM
```

the variable X acquires the value of the string "A PROBLEM". You may prove this very simply. First enter READ X, press the entry key, and then give the value you wish to assign to X. Next use the WRITE command to display X's value, by typing:

```
WRITE X
```

and the terminal replies by displaying:

```
A PROBLEM
```

Because you knew that the READ command expected a reply from you, you entered a value for the variable X. Remember that a user of your code, who is trying to enter data for you, needs prompting in some way as to when he should respond. You can prompt him very easily by preceding any variable name with an explanatory string of characters. For example, instead of entering:

```
READ X
```

you could enter:

```
WRITE "WHAT DO YOU SEE?" READ X
```

or you could have used only the READ command with two arguments:

```
READ "WHAT DO YOU SEE? ",X
```

The terminal would then display:

```
WHAT DO YOU SEE?
```

After you enter your reply, the total display would show:

```
WHAT DO YOU SEE? A PROBLEM
```

Again, the value "A PROBLEM" is assigned to the variable X.

Note the space between the question mark of the prompt and the quotation mark ending the prompt. Such a space is helpful for clarifying where the prompt ends and the user's response begins. Unlike the required space after each command, the space in the end of a prompt is not essential, but you will usually find that your users like it.

Try some code yourself. Use your own variable names and your own prompting text if desired. Enter any values you like. Check that things have gone correctly by using the WRITE command to inspect the value of each variable.

1.6. SET

The SET command allows you to assign a value to a variable. For example, we supposed in Section 1.4 that NAME had been assigned the value "JOAN". We could have made this assignment by entering the code:

```
SET NAME="JOAN"
```

The SET command is followed by one space. (Yes, you are getting the general picture: usually each command is follwed by a single space.) Next comes a variable name (in this case, NAME), followed by the equals symbol (=) that separates the variable from the value being assigned, and the value ("JOAN") that the variable NAME is to take.

How would you assign to the variable WHY the value of "WHY IS THIS PATIENT HERE"? It could be assigned by the following code:

```
SET WHY="WHY IS THIS PATIENT HERE?"
```

Alternatively, you may give the variable WHY such a value by means of the READ command, in such code as:

```
READ "QUESTION: ",WHY
```

Try some SET examples yourself. What are the results of the following:

```
SET A="HELLO"

SET B=36+41

SET C="A"

SET D=A
```

The answers that you will obtain when you write out the values are:

WRITE A	gives	HELLO
WRITE B	gives	77
WRITE C	gives	A
WRITE D	gives	HELLO

The last one is particularly interesting: not only may you assign to a variable a value that is some string of letters of a number, but you may also assign a value of some other variable.

SET, like many other commands, may take more than one argument. For example, the above four variables could be assigned values by:

```
SET A="HELLO",B=36+41,C="A",D=A
```

1.7. KILL

When you have finished with any variable, you may get rid of it so that it does not remain to confuse you later. The only way to get rid of a variable is to KILL it. To get rid of variable A you enter:

```
KILL A
```

For many versions of MUMPS you can verify that A is no longer defined by typing:

```
WRITE A
```

because many (although not all) computers reply with an error message indicating that A is no longer defined.

To remove several variables, simply list the variable names separated by commas after the KILL command; do not put spaces within the argument. For example:

```
KILL A,B,C,D
```

causes the elimination of the variables just defined in Section 1.6 by means of the SET command. Again, you may confirm this elimination by trying to WRITE any variables that you have KILLed: the computer may tell you that there are no longer available.

1.8. AVOIDING AN ARGUMENT

Some commands do not have to be followed by an argument. In particular, the KILL command without arguments eliminates all defined local variables. Thus, if you enter:

```
KILL
```

all of your local variables are removed. This may be confirmed by subsequently trying to WRITE the value of any variable defined before the KILL command was given. The computer may respond with a message indicating that the specified variable is no longer defined.

Note that the syntax, or grammatical structure, of MUMPS means that each command is followed by one blank space, an argument, and at least one more blank space. Then the next command begins, and so on. In the situation where no argument is specified, the two blank spaces are no longer separated by a visible argument. Therefore the command appears to be followed by two blank spaces and then the next command. For example, to KILL all variables and assign a value to a new variable A, enter:

```
KILL  SET A=-1
```

not:

```
KILL SET A=-1
```

1.9. EXERCISES

Answers are given in Appendix 3.

1.1 What is the difference between a command and an argument?

1.2 What is the difference between a constant and a variable?

1.3 What is the difference between:

WRITE X

and

READ X

1.4. What results from you giving the following MUMPS code to the computer:

(a) WRITE "PATIENT NAME"

(b) READ X

(c) READ "PATIENT NAME: ",X

(d) KILL
WRITE A

(e) SET %=A/B*100 WRITE %

1.5. Write MUMPS code to do the following:

(a) Ask the user to enter his name and assign that value to variable CALL.

(b) Remove any defined variables and then define variable B as having the value 2 and variable SPORT as having the value "RUNNING".

CHAPTER 2

WRITING A FIRST ROUTINE

2.0 INTRODUCTION

In Section 2.1, a problem to be encoded is introduced. It concerns a calculation of the individual benefits of exercise. Some new features of MUMPS are needed to solve this problem, particularly the IF command (reviewed in Section 2.2) and arithmetic operators (Section 2.3). These features lead to the concept of a "truth value" and the use of the negation operator to reverse a truth value (Section 2.4). MUMPS code may be executed immediately or stored for execution when needed, as described in Section 2.5. The GOTO command allows transfer of execution from one part of MUMPS code to another, as described in Section 2.6. The QUIT command (Section 2.7) ends the execution of code and the DO command (Section 2.8) allows temporary transfer of execution.

All of the MUMPS components necessary to solve the exercise program have now been presented, and a sample solution is shown in Section 2.9. Answers to various questions asked in this chapter are given in Appendix 3.

2.1. MUMPS EXAMPLE: A METHOD OF CALCULATING THE INDIVIDUAL BENEFITS OF EXERCISE

For the first example that we solve by writing a sequence of MUMPS instructions, consider the calculation of the benefits of exercise. Dr. Kenneth H. Cooper has published two books, Aerobics in 1968 and The New Aerobics in 1970, that discuss the benefits to the heart and lungs of continuous exercise. Aerobics is exercise that lasts long enough to cause a considerable increase in your pulse rate and your oxygen consumption. Examples of aerobic exercise are walking or running, swimming, and cycling, when performed continuously for at least 12 minutes. The further you run, swim, or bicycle, and the faster your rate of exercise, the more aerobic exercise you give your body. Dr. Cooper has developed a scoring system for the amount of aerobic exercise accomplished. For example, if you run 1 mile in 9 minutes you score 4 points, but if you can do it in 7 minutes you score 5 points. However, if it takes you 20 minutes or more, you have taken too long for the exercise to have an aerobic effect and you score no points. As you increase your distance, you get more points, even if you are quite slow. A reasonable goal for a week is a total of 30 points. This could be achieved by running 1.5 miles in about 14 minutes on 5 days out of the week, or in many other ways. For more facts and ideas, see Dr. Cooper's books, published in paperback by Bantam.

A sample interaction with a MUMPS computer program to calculate exercise points is shown in Figure 2.1. The computer asks the user to enter the distance that he has run. If he says that he has run an unreasonably large distance, the computer refuses to accept it and asks for a more plausible value. When a reasonable distance has been entered, the computer asks for the time taken to cover the distance. If the time taken means that the person has gone at an impossible rate, the computer will not accept that and asks the user to specify his distance and time again. Ultimately, the user enters a reasonable distance run and time taken, and the computer calculates the earned number of aerobic-exercise points.

```
HOW MANY MILES DID YOU RUN?  200
IMPOSSIBLE
HOW MANY MILES DID YOU RUN?  7
  VERY GOOD
HOW MANY MINUTES DID THAT TAKE?  30
THAT'S FAST!
YOU JUST ADDED 42 POINTS AND YOUR TOTAL IS 42
ENTER MORE

HOW MANY MILES DID YOU RUN?  7
  VERY GOOD
HOW MANY MINUTES DID THAT TAKE?  4
THAT'S FAST!
ACTUALLY, THAT'S IMPOSSIBLE.....START AGAIN

HOW MANY MILES DID YOU RUN?  7
  VERY GOOD
HOW MANY MINUTES DID THAT TAKE?  49
THAT'S FAST!
YOU JUST ADDED 35 POINTS AND YOUR TOTAL IS 77
ENTER MORE

HOW MANY MILES DID YOU RUN?  3
  VERY GOOD
HOW MANY MINUTES DID THAT TAKE?  27
YOU JUST ADDED 12 POINTS AND YOUR TOTAL IS 89
ENTER MORE

HOW MANY MILES DID YOU RUN?  1
HOW MANY MINUTES DID THAT TAKE?  10
YOU JUST ADDED 3 POINTS AND YOUR TOTAL IS 92
ENTER MORE

HOW MANY MILES DID YOU RUN?

>
```

Figure 2.1. Sample interaction between a user and a computer program to calculate points for exercise. Note that the computer refuses to accept excessively long distances claimed or high speeds. The points added increase with distance and speed.

Before you look at the MUMPS code that performs the interactions shown in Figure 2.1 and that calculates your exercise points, you need to learn the new features of MUMPS that are presented in the rest of this chapter. The sample of code is shown in the last section.

2.2. INTRODUCTION TO THE IF COMMAND

A major way in which you can use a computer is for making decisions depending upon the data entered by a user. For example, suppose you were looking for a person whose name is "DAVID". You ask each user for his name, thus:

```
READ "NAME PLEASE: ",X IF X="DAVID" WRITE "HELLO, DAVID.  I HAVE
BEEN WAITING FOR YOU"
```

Whatever the user enters as his name is assigned as the value to the variable X. Using the IF command, you can then inspect the variable X by testing whether its value is equal to the string "DAVID". In the example, the IF command appears after a value is read in for the variable X. The IF command is followed by its argument, thus:

```
IF X="DAVID"
```

Whenever the argument of an IF command is true, any code that follows it on the same line is executed. Otherwise the rest of the line is ignored. So, if the user claims that his name is DAVID, the computer greets him by printing out the text following the WRITE command. Alternatively, if his name is not DAVID, the greeting is not given.

There are two points to note about the example. First, observe that the equals sign ("=") is used to check whether the variable X has the value of "DAVID". A similar example is the checking of whether a variable AGE is equal to 21, by writing:

```
IF AGE=21
```

Now you try.

Question 2.1. What would you write to check whether the variable SHIP has the value "TITANIC"? (After you have written down your answer, you many look at Chapter 2's answers in Appendix 3.)

The second point illustrated by the example is that a series of commands and their arguments may appear on a single line. You will usually find it helpful to you in remembering later what your code does if you put together on the same line those commands that are in some way on the same topic. In the example at the start of this section, the three commands were concerned with obtaining a value for the variable X (READ), inspecting its value (IF), and taking some action depending upon what the value was (WRITE). Be careful not to put too many commands on a line, however, as most systems will have a limit to the length of a line of code (usually 256 characters).

The IF command, like the SET command, may take multiple arguments. For example:

```
IF A=3,B=5 SET C="PHARMACY"
```

tests for the values of both A and B, however, it is important to remember that the value of B will not be checked unless the value of A is 3.

2.3. ARITHMETIC OPERATORS: THE COMMON ONES

You are probably familiar with the arithmetic operators of addition and subtraction, represented by + and - respectively. In MUMPS, just as in real life, it is true that 1+1 equals 2, and that 7-3=4. Two other common arithmetic operators are multiplication and division. In MUMPS the symbol * is used for multiplication, to avoid confusion with the letter x. For example, 8*5 has the value of 40. Division is represented by the symbol /. You may tell your MUMPS computer to:

```
WRITE 96/12
```

The answer should be 8.

There are two other operators that you may know already. These represent "less than" and "greater than". Suppose you want to find whether some variable called MILES, say, has a value of less than 2. You use the less-than operator, which is the "<" symbol. You also use the IF command introduced in the last section. The result is:

```
IF MILES<2
```

This simply means "if the value of the variable MILES is less than 2". When MILES has a smaller value than 2, any commands occurring on the same line after the IF command will be executed. Alternatively, when the value of MILES is 2 or more, MUMPS ignores the instructions that follow IF on the same line, and continues with the instruction that starts the next line.

If you have not seen the less-than operator before, notice that its narrow end points leftward to the expression that is being inspected to see whether it is smaller than the expression on the right. For the greater-than operator, on the other hand, its wider end points leftward to the expression that is being inspected to see whether it is greater than the expression to the right. For example, when you want to check whether the value of MILES is greater than 100, you put:

```
IF MILES>100 WRITE "IMPOSSIBLE"
```

Here, if the user claims to have run more than 100 miles on a single day, the computer expresses disbelief.

Question 2.2. If the variable SUM has the value 97, is it true or false that SUM<40?

2.4. TRUTH VALUES AND THE NEGATION OPERATOR

The argument of an IF command may be either true or false. A value of true or false is called a "truth value". Truth values are used by various components of the MUMPS language, but here we deal only with the explicit argument of an IF command. When an expression is true, it has a truth value of 1, and when an expression is false it has a truth value of 0. For example, in the MUMPS code:

```
IF AGE=100 WRITE "CONGRATULATION"
```

the argument of the IF command is AGE=100. When AGE has the value of 100, the truth value of the expression AGE=100 is 1, and the WRITE command is executed. If the value of AGE is not 100, the expression AGE=100 is false and its truth value is 0 (zero). In this case, the WRITE command is not executed.

Question 2.3. What is the result of the following code when NAME has the value "AWFUL":

```
IF NAME="AWFUL" READ "WOULD YOU LIKE TO CHANGE YOUR NAME? ",X
```

And what happens when NAME has the value "NOT AWFUL"?

In some situations it may be more appropriate to consider what a variable does not equal rather than what it does equal. For example, suppose we know that the value of the variable POINTS

is a positive, nonzero number; we want to write out the number, followed by the word "POINT" if the number is exactly 1, but by the plural word "POINTS" if the number is any other value such as 0.5 or 77. The following code will do that:

```
WRITE POINTS," POINT" IF POINTS'=1 WRITE "S"
```

When POINTS has any value other than 1, the WRITE command that follows the IF command is obeyed and the letter S is added.

The negation operator may be used in conjunction with any arithmetic relational operator. Specifically, it may be associated with the "less than" (<) and "greater than" (>) operators. For example:

```
IF TIME'<8
```

checks whether the value of variable TIME (perhaps measuring seconds or minutes) is not less than 8. The use of the negation operator here allows a more precise statement of the condition on the value of TIME than if it were expressed as:

```
IF TIME>7
```

or

```
IF TIME>7.99
```

Question 2.4. What is the argument of the IF command that will check whether the value of the variable HEIGHT is not more than 60?

Remember that the IF command may have more than one argument, as may most of the other MUMPS commands. For example, if you want X to have the value "N" and the VALUE to exceed 10, enter:

```
IF X="N",VALUE>10...
```

2.5. HOW TO DEFER EXECUTION OF YOUR CODE

So far, the examples that have been shown and that you should have been trying out for yourself are executed by MUMPS immediately you have typed the appropriate characters and pressed the entry key. When the code you write is executed immediately, you are said to be working in "direct" mode or "programmer" mode.

While it is essential to be able to use "programmer" mode and to test the various features of MUMPS directly, it is also necessary to be able to store instructions so that they can be executed later. The way in which you instruct a MUMPS computer that a line of code is to be stored for later execution varies

between systems. In many systems you hold down the control key and press the letter I. All code that you enter from this point until you press the entry key is stored as a line of code for subsequent execution. Whatever commands (WRITE, SET, etc.) you enter after the "deferred execution" or "line start" indicator are not executed immediately, but saved for later use in "indirect" or "user" mode. You must again use the control key and the letter I (or whatever may be your combination of keys) at the start of each line that is to be stored. In other systems, you may have to use a special "editor" in order to store and modify code; there you need invoke the editor only once and all code written under it will be stored until you leave the editor and return to direct mode. Write down here, for your future reference, the key(s) that you press on your system to tell MUMPS that your code is to be stored rather than executed; indicate whether you press these keys for every line, or just once to invoke an editor for many lines.

You will find it helpful to give a name or label to several of the lines of code that you create. The label name may have a similar structure to the name of a variable in that it may begin with an alphabetic or the "%" symbol and may continue with up to seven alphabetics and/or numerics, making a total of up to eight characters in all. In addition, a line label may be composed entirely of numerics. You may think of a line label as being like the start of a new paragraph in English, to indicate when the topic changes: a label is useful for flagging a change of purpose or emphasis in the code. If mnemonic labels are used, the labels should not only document that a change in emphasis is occurring but also indicate the nature of the change. NEXT is a valid label. So is %A. So is 13.

In systems where you indicate that you are writing a line to be stored by using the control key plus letter I, or some other combination of keys for each line, you add a label to the line as follows. Write the letters in the label; press the keys that are necessary to tell the system to store the code; enter the line of code. If your procedure is different, write it here:

A collection of lines of code is called a "routine". The code that comprises a routine may be given a name and stored on your system's disc or other mass-storage equipment. It may also be retrieved at a later time. Possible routine names are the same as line labels. Some examples of storing and retrieving routines are given in Section 3.1 and 3.2 and in Appendix 2.

2.6. THE GOTO COMMAND FOR THE PERMANENT TRANSFER OF EXECUTION

GOTO is a command that allows the programmer to branch from executing code at one part of the routine to some other code. For example, suppose the programmer had written the following three lines of code:

```
AAA   WRITE "THIS IS THE BEGINNING" GOTO XYZ WRITE "NO"
AAB   WRITE "YOU WILL NEVER SEE THIS WRITTEN" QUIT
XYZ   WRITE " AND THE END."
```

Note that each of these lines has a label, the first line being labelled AAA, the second AAB, and the third XYZ: the particular MUMPS implementation leaves a gap of three spaces between each label and the code on the line it labels, so that the code may stand out more clearly. This display produced by this code is:

```
THIS IS THE BEGINNING AND THE END.
```

"NO" is never written because the GOTO command on line AAA causes the last command there (WRITE) to be avoided. The line labelled AAB is never executed either because the instruction GOTO XYZ in line AAA causes the line AAB to be bypassed. If the GOTO command in line AAA were to have AAB as its argument, line AAB would be executed and line XYZ would not because of the QUIT command at the end of line AAB. Alternatively, if the instruction is GOTO AAA, the words "THIS IS THE BEGINNING" would be written out endlessly.

In addition to being able to alter the flow of control within a routine by causing branching to different lines, the GOTO command may also be used for transferring execution from one routine to another. Such transfer is done by prefixing the argument of the GOTO command with a circumflex ("^"). To illustrate this, look at the following example. Suppose we have a routine named ALPHA that contains the following lines of code:

```
AAA   WRITE " THIS IS THE BEGINNING"
XYZ   WRITE " AND THE END."
```

and suppose we have another routine named OVE which is simply:

```
OVE   KILL  WRITE "START." GOTO ^ALPHA WRITE " THE REAL END."
```

Suppose routine OVE is being executed first. All local variables are eliminated by the argumentless KILL command. Then the following is written:

```
START.  THIS IS THE BEGINNING AND THE END.
```

Note that the GOTO command starts execution of routine ALPHA at

its first line. Furthermore, execution does not return to routine OVE when routine ALPHA is completed. Therefore the instruction WRITE " THE REAL END." in routine OVE is not executed.

The programmer wants his code to branch most often to a particular line of the same routine or to the first line of some other routine as shown in the above examples. However, he may want it to branch to some particular line after the first one in a different routine. This is accomplished by putting as the argument of the GOTO command:

1. The label of the desired line of the new routine.

2. The circumflex (^) symbol to show that a different routine is being invoked.

3. The name of the routine into which the programmer wishes to branch

For example, to cause branching to line A7 of routine COLONY, write the following code:

```
GOTO A7^COLONY
```

Note that after the GOTO command you specify first the particular line (A7) to which execution is to branch. Then you put the circumflex, signifying that you are wanting line A7 in some routine other than the one that is being executed. Finally you specify the name of the other routine.

Consider these further examples.

```
GOTO LINE
```

causes execution to branch to the line of code labelled LINE in the present routine. Alternatively:

```
GOTO LINE^ROUTE
```

causes execution to branch to the line labelled LINE but appearing in the routine named ROUTE. Finally

```
GOTO ^ROUTE
```

does not specify explicitly the line of routine ROUTE at which execution is to begin, but in practice it causes execution to branch to the <u>first</u> line of ROUTE.

Question 2.5 Using the GOTO command, give the instructions for branching from executing code in some routine to executing code in (a) a different routine named SUNSHINE, starting at the first line, (b) a different routine named SUNSHINE, starting at

the line labelled 1, (c) line EGGS of the present routine, and (d) line % of routine CATASTRO.

2.7. THE QUIT COMMAND

The QUIT command causes the current execution of instructions to end. For the time being, the QUIT will be used without an argument, and is therefore followed by at least two spaces (see Section 1.8) to separate it from any command that follows it. For example, if you enter:

```
SET X=1 WRITE X QUIT  WRITE " HERE BY WITCHCRAFT"
```

the computer writes

```
1
```

and nothing else. The QUIT command stops execution at the point where the QUIT is encountered, and nothing further is executed. The QUIT command will be discussed in the next section on the DO command in which context it is most commonly used.

2.8. THE DO COMMAND FOR THE TEMPORARY TRANSFER OF EXECUTION

The DO command is similar to the GOTO command in that the DO command allows you to start executing code at some line other than that currently being executed. However, the transfer of execution is temporary. When a QUIT command is encountered, indicating completion of the code whose execution the DO command started, MUMPS returns to the code following the DO command and starts to execute it. For example, suppose MUMPS is executing the following routine:

```
AAA   WRITE "THIS IS THE BEGINNING" DO XYZ WRITE " YES,"
AAB   WRITE " THIS IS VISIBLE." QUIT
XYZ   WRITE " AND THE END."
ZZZ   QUIT
```

then the following display is produced:

```
THIS IS THE BEGINNING AND THE END.  YES, THIS IS VISIBLE.
```

Execution begins at the first line, AAA, where the WRITE command is obeyed. The DO command causes the line XYZ to be executed, and all lines following XYZ until the QUIT command in line ZZZ is encountered. Execution then returns to the next instruction following DO XYZ, i.e., WRITE " YES,". Execution continues until the next QUIT command is encountered, at the end of line AAB, at

which point no more code in this routine is executed. Note that the lines XYZ to ZZZ form a "subroutine" or subsection of the main routine.

Another example of the DO command is

```
DO 34 SET A=-1
```

It causes the code starting at the line labelled 34 to be executed. The code from line label 34 until a QUIT command is encountered forms a "subroutine". It continues to be executed until it is ended by the QUIT command. When a QUIT has been encountered, the next command after DO 34 is executed thereby assigning a value of -1 to the variable A. It is wise to avoid use of a GOTO command to branch away from the code of the subroutine. Otherwise a QUIT command eventually occurs that has the effect of ending the subroutine and, often surprisingly, returning to the next command after the original DO command.

The DO command may be used for branching not only within a routine but also between routines. For example, suppose that we have the routine:

```
CAL   KILL  WRITE "CALLING.  "DO ^AAA WRITE " RETURNED."
      SET B=3
      QUIT
```

Note the circumflex before the name of routine AAA. The circumflex shows that the DO command invokes routine AAA (which appeared earlier in this section). The resulting display when line CAL is executed is:

```
CALLING.  THIS IS THE BEGINNING AND THE END.  YES, THIS IS
VISIBLE.  RETURNED.
```

Observe that the QUIT command in line AAB of routine AAA causes MUMPS to return to executing code in line CAL. On the other hand, the QUIT command in line ZZZ of routine AAA causes the subroutine starting at line XYZ to be ended; in that case execution returns to line AAA after the instruction DO XYZ that invoked subroutine XYZ.

A DO command, then, may invoke execution of a subroutine that starts at a line within the present routine. For example:

```
DO L
```

starts execution of the line labelled L. When a QUIT command is encountered execution of the subroutine ends and the next command after DO L is executed. A DO command may also invoke execution of another routine. For example:

```
DO ^R
```

causes the routine named R to be executed starting at its first line. When a QUIT command is encountered, the next command after DO ^R is executed.

Finally, the DO command may instruct that some other routine is invoked starting at a specified line. For example:

```
DO L^R
```

causes the routine named R to be loaded and its code to be executed starting at line L. When a QUIT command is encountered, the next command after DO L^R is executed.

2.9. MUMPS EXAMPLE: SOLUTION

A sample solution for the aerobics exercise problem is shown in Figure 2.2. Code is shown only for running, but similar code could be written for swimming and for cycling.

In the MUMPS implementation used, the label of each line that has one is printed on the left. The code for every line is indented by six characters from the left margin. The first line is labelled ZEX2. It has a semicolon at the position where a command would appear. A semicolon in the position of a command is used in MUMPS to specify that whatever follows the semicolon is not to be executed; instead the characters after the semicolon are ignored and they serve only as an explanation of the purpose of the code.

In line ZEX2+1 (that is, the line immediately after ZEX2) all local variables are removed. The variable SUM, which is to accumulate the total points, is initialized to zero.

```
ZWX2    ;ZIM;28 AUG 79; COOPER'S AEROBICS POINT SYSTEM
        (1970) KILL  SET SUM=0

RUN     READ "HOW MANY MILES DID YOU RUN? ",MILES,! IF MILES=0 GOTO SWIM
        IF MILES>100 WRITE "UNLIKELY",! GOTO RUN
        IF MILES>2 WRITE " VERY GOOD",!
        READ "HOW MANY MINUTES DID THAT TAKE? ",MINS,!
        IF MINS=0 DO ZERO GOTO RUN
        SET T=MIN/MILES IF T<8 WRITE "THAT'S FAST!",!
        IF T<3.7 WRITE "IMPOSSIBLY FAST .... START AGAIN",!! GOTO RUN
        IF MILES<2,T>19.99 SET POINTS=0 GOTO ERUN
        IF T<6.5 SET POINTS=6 GOTO MILES
        IF T<8 SET POINTS=5 GOTO MILES
        IF T<10 SET POINTS=4 GOTO MILES
        IF T<12 SET POINTS=3 GOTO MILES
        IF T<14.4 SET POINTS=2 GOTO MILES
        IF T<20 SET POINTS=1 GOTO MILES
        SET POINTS=0.5

MILES   SET POINTS=POINTS*MILES

ERUN    DO SET GOTO RUN

SWIM    ;CODE TO BE WRITTEN

SET     SET SUM=SUM+POINTS
        WRITE "YOU JUST ADDED ",POINTS," POINT" IF POINTS'=1 WRITE "S"
        WRITE " AND YOUR TOTAL IS ",SUM,!,"ENTER MORE",!! QUIT

ZERO    WRITE "YOU CAN'T DO THINGS IN NO TIME",!! QUIT
```

FIGURE 2.2. MUMPS code to calculate aerobics points and give the interaction shown in Figure 2.1. The exclamation mark in lines RUN, RUN+1, and several others causes a new line to be started for the display of characters; this is discussed in chapter 7

Line RUN requests from the user the number of miles he has run and assigns the value he specifies to the variables MILES. If the user has run no distance, the code branches to the line labelled SWIM where execution of this routine is ended by the QUIT command.

Lines RUN+1 and RUN+2 check the value of MILES. Line RUN+3 requests the user to specify the number of minutes he took to cover the distance he stated. This time is assigned to the variable MINS. The next line checks whether the value of MINS is zero and the following line calculates a value denoting the speed in minutes per mile and assigns this to the variable T. The value of T is checked for plausibility. Finally line RUN+7 to line MILES calculates the number of points earned and assigns them to the local variable POINTS.

Line ERUN ends the calculation of points for running by involving the subroutine that starts at line SET. There the value of SUM is incremented by the value of POINTS. After completion of the subroutine at the QUIT command in line SET+2, execution of the code returns to line ERUN. It then branches back to line RUN, allowing the entry of any additional distance run. In this way, the user may continue to accumulate points until he has specified all of his running and finally enters a value of zero miles.

CHAPTER 3

MORE ON HOW TO PROGRAM

3.0. INTRODUCTION

Standard MUMPS does not specify explicitly the procedures for storing, retrieving, and displaying code. Therefore each MUMPS implementation may have different procedures for performing these operations. Chapter 3 contains examples of such non-Standard features and space where you may write the versions that you use. In MUMPS the initial letter of Z is reserved for non-Standard commands.

Section 3.1 shows ways of storing MUMPS routines on disc or other long-term storage devices. Each programmer may be thought of as having his own partition or working space in which he develops his code before he stores it. He can retrieve code from long-term storage into his working space as shown in Section 3.2. Code may be displayed or printed as illustrated in Section 3.3. Lines of code may be deleted and new lines added as shown in Sections 3.4 and 3.5. Use of the BREAK command and some ways of editing code are discussed in Section 3.6.

The importance of dividing code into logical and relatively small modules is noted in Section 3.7. Finally Section 3.8 presents the HALT command for terminating use of a workspace.

3.1. STORAGE OF MUMPS ROUTINES

In Chapter 2 you saw how to write lines of code that were not executed directly but were saved in your work space for subsequent, indirect execution. Remember that such a sequence of code lines is called a routine.

So that you do not have to rewrite your routine each time it is needed, MUMPS implementations provide the capability to copy any routine from a work space on to a long-term storage device. Most commonly such a device is a disc, but the storage procedure used should be simple and probably independent of the storage device. It is fairly common for a routine to be stored by means of a command. Names that have been used for this command include ZSTORE, ZSAVE, and ZFILE. The command's argument is the name being assigned to the routine. A name is assigned to a routine the first time it is stored. A routine name may have the same format as a label name (see Section 2.5). Consequently a routine name may contain alphabetics and numerics. It may also begin with the percent symbol (%). No more than eight characters should be used for routine names, to be consistent with the eight-character

limit for differentiated names. Examples of suitable routine names are ROUTINE, %ROUTINE, ROUTINE7, and R. If the command for storing a routine is ZSTORE, then the first routine would be copied on to the long-term storage device and stored there as a result of:

ZSTORE ROUTINE

Alternatively if ZFILE is the command, then the second routine is stored by:

ZFILE %ROUTINE

If your implementation lets you store a routine in this way, by means of a command followed by a routine name, write here how you would store a routine with the name TEST:

MUMPS implementations may use other means to store a routine. If this is the case for yours, write here the interaction that is necessary for you to store a routine named TEST:

3.2. RETRIEVAL OF CODE

After you have written a routine and stored it, you can work with other routines. At some point you may wish to retrieve the first routine from long-term storage, perhaps in order to modify a line of code. It is common for a routine to be retrieved by means of a command whose argument is the name assigned to the routine. The command could be ZLOAD or ZRETRIEVE, which are shown retrieving ROUTINE7 and R respectively in these examples:

ZLOAD ROUTINE7 and

ZRETRIEVE R

Be particularly careful if you abbreviate your commands (as most MUMPS programmers do): you should not accidentally use some of the commands suggested in this chapter instead of those appropriate to your MUMPS implementation. In particular, abbreviation of ZRETRIEVE to ZR should be used cautiously in case it causes code to be REMOVED instead of RETRIEVED.

If your implementation lets you retrieve a routine by means of a MUMPS command followed by a routine name, write here how you

would copy the routine named TEST from long-term storage into your working space:

A MUMPS implementation may use other means than such non-Standard MUMPS commands to retrieve a routine. If this is the case for your implementation, write here how you would retrieve a routine with the name TEST:

3.3. PRINTING A ROUTINE

So that you can inspect the code you have written, MUMPS implementations allow you to print your lines of code. Many implementations use a command such as ZPRINT, ZLIST, or ZDISPLAY. The use of such a command without an argument may cause all lines of code in the work space to appear. Accordingly, if the routine shown in Figure 2.2 is in the work space, the use of the appropriate command such as:

```
ZPRINT
```

results in the display of all the code. Write here the instruction(s) that you have to give to display all lines of code in your work space:

Selected lines may often be displayed by using line labels as arguments. For example the first and third lines of the routine in Figure 2.2 might be displayed by:

```
ZPRINT ZEX2,RUN
```

or

```
ZLIST ZEX2,RUN
```

An unlabeled line may be displayed by specifying the distance it occurs below some labelled line. For example the fourth line in Figure 2.2 is one below line RUN and so may be referred to as line RUN+1. It could be displayed by:

```
ZPRINT RUN+1
```

Question 3.1. If your system used the ZPRINT command, how would you display the code in the line labelled MILES and the line that is two below the line labelled SET?

Some implementations provide a capability to display a sequence of lines by specifying the first and last lines of the sequence required. The two lines specifications are separated by a colon instead of a comma. (If a comma is used, the first and last lines are printed but not the lines between.) For example, to display the sequence of lines from RUN to MILES you might write:

```
ZPRINT RUN:MILES
```

and to print those lines followed by lines ZEX2 and then the line one below line SET, you might write:

```
ZPRINT RUN:MILES,ZEX2,SET+1
```

Specify the way you would display a sequence of lines followed by discrete lines, using in your example the lines just named (namely from RUN to MILES, then ZEX2 and SET+1):

Lines to be printed may also be specified based on how far they are from the start of the routine. For example:

```
ZPRINT +1
```

could cause the first line to be printed and:

```
ZPRINT +7
```

might cause display of the seventh line.

3.4. DELETION OF LINES OF CODE

If a routine is unsatisfactory you may wish to delete it or remove it from your work space. Your MUMPS implementation may include a command such as ZDELETE or ZREMOVE to allow you to do this. For example:

```
ZDELETE
```

may remove all your code. Write here what you must specify for your implementation in order to remove all lines of code in your work space:

It is usually possible to delete specified lines, by naming the lines separated by commas. For example if lines labelled A and % are to be deleted, the instruction given might be:

```
ZREMOVE A,%
```

Specify here how you would get your implementation to delete those two lines:

Alternatively you may delete lines that are a certain number of lines below certain others. For instance, to delete the first and second lines below line BAT, you might write:

```
ZREMOVE BAT+2,BAT+1
```

(Note that if BAT+1 is specified before BAT+2, then the line BAT+3 is deleted instead of BAT+2.) Specify how you would get your implementation to delete those lines:

You may also delete lines based on how far they are from the start of the routine. For example you might delete the sixth line by:

```
ZREMOVE +6
```

Specify here how you would get your implementation to delete that line:

Just as some implementations allow a sequence of lines to be printed by specifying the first and last lines with a colon between them, so some implementations allow a sequence of lines to

be deleted. For example, to delete lines from XA to XM, you might enter:

ZREMOVE XA:XM

Specify here how you make your implementation delete lines from XA to XM:

3.5. INSERTION OF LINES

When you have a routine in your work space, you may add lines to it by writing code as discussed in Chapter 2. The only problem that you may have is to ensure each new line is added at the appropriate place. The location of a new line depends upon what was being done previously. For many implementations if a new line is displayed or printed, then the next line created is placed immediately after that line. In some implementations there is a special command (such as ZMOVE) that allows you to specify the line after which you wish to insert the new code that you write next. Specify here the way that your implementation lets you choose where you will place the next line of code that you add:

3.6. BREAK COMMAND AND CODE EDITING

There are two main approaches to identifying errors in written code. First read it carefully to check that what you have written should perform the actions you want. Secondly test the code by executing it for typical data.

The BREAK command extends your capability to test your code. This command may be placed in any line at any position where a command can appear. An example is:

```
READ "ANSWER ",ANS SET L=A+$LENGTH(ANS) BREAK  WRITE "CONTINUING"
```

In this case, the READ and SET commands are executed and then the BREAK command is encountered. BREAK causes a pause in the execution of the stored code. At this time, the programmer may give direct-mode instructions, particularly to test which variables are defined and what their values are. In many implementations, the user is told not only that a BREAK command has been encountered, but also in which lines and perhaps even where in the line that command occurred. Thus guided, the programmer may identify which of several BREAK commands is being executed and the sequence in which the BREAK commands are reached.

If a located error extends over much of a line, it may be simplest to erase that line (see Section 3.4) and write a new one (Section 3.5). Alternatively part of a line may be changed with the aid of an editor on your implementation. Examples of how an editor can be invoked are:

```
DO ^%E
```

and

```
XECUTE ^%
```

(The DO command was introduced in Chapter 2; details of the XECUTE command are not needed until Chapter 12.) On some implementations you may already be in editing mode so that you do not invoke an editor explicitly. If you can invoke an editor, write here how you do so:

Once you have invoked an editor, your interaction with it depends upon this particular editor. A sample interaction is shown in Figure 3.1 where the user (whose responses are underlined) is prompted by:

```
L:
```

for the label of the line to be edited. First line X is specified and the contents of this line are displayed. The user is prompted by:

```
R:
```

for the characters to be replaced. The part of the line up to the change is shown, the user enters the correction (which is underlined here purely for clearer illustration of what the user types; the underlining does not appear on the computer terminal), and the computer completes the line. The user may specify more characters to change or may work with another line as illustrated in the rest of the figure. In the space below specify similar information for the MUMPS implementation you use:

DO ^%E	User invokes editor. (User's input is underlined.)
L: MILES	User specifies that line MILES will be edited.
MILES SET POINTS=POINTS+MILES	Editor displays line MILES.
R: PI	User specifies characters to change.
MILES SET POINTS=POINTS+MILES	Editor displays characters up
R: +	to those to be changed; user
MILES SET POINTS=POINTS*MILES	enters new characters; editor completes line.
R: _	User has no more changes to make.
L:ZEX2+3	User specifies next line to edit.
.	Editing continues
.	.
.	.
.	.
.	.
L:_	User has no further changes to make.

Figure 3.1. Sample interaction with a hypothetical code editor.

3.7. DIVISION OF CODE

Standard MUMPS recommends that the space occupied by any routine plus all local variables and temporary results co-existing with the routine be limited to 4000 characters. You may be using a MUMPS implementation that limits the characters to fewer than this. If you try to fit more code and data into the space than there is room for, you will probably be given an error message. It will not be possible to use the code until you have removed some lines or killed some variables. To reduce the possibility that such a problem with storage space occurs, many programmers impose a limit for themselves on the maximum size of a routine. For example, you might plan to have routines of about 1000 characters, with an upper limit of 2000 characters. In Figure 2.2 there are 24 lines of code. The first line, labeled ZEX2, shows 4 characters for the label and 52 for the MUMPS code (which here is a comment). Implementations vary in the number of characters they use for start-of-line and end-of-lne indicators, but as many as four additional characters per line may be needed. The code of Figure 2.2 occupies about 1000 characters or an average of about 40 characters per line. In practice, a routine size like this is convenient to look at and think about.

A routine should not only be smaller than some maximum size. It should also have logical significance so that an activity is begun and completed within it. This allows each routine to be thought of as a building block. These blocks are linked using the DO and (with caution) GOTO commands (see Chapter 2) to construct the application.

Duplication of code may be reduced by identifying each task needed at more than one place. The code for such a task may be made into a single routine that can be invoked from different places. This simplifies the creation and modification of the code because there is only one version to be dealt with.

3.8. HALT COMMAND

When programming and testing of code is completed for the moment, give the HALT command. This command does not take an argument. It terminates your programming and releases for other users the part of the computer and any other resources that have been under your control. Among the resources released is the space where you are working (your "workspace"). Do not worry. You will get a new one next time you sign on to MUMPS.

You may put the HALT command in your code if you wish, so that execution of your application ceases when a particular point is reached, such as when a user of your application gives a particular answer. For example:

```
READ "WANT TO CONTINUE? ",X IF X="NO" HALT
```

causes execution of the HALT command and termination of execution if the user answers NO. Note how much more the HALT command does than the QUIT command.

CHAPTER 4

FUNCTIONS

4.0. INTRODUCTION

MUMPS contains a number of functions that can be used to manipulate data. This chapter introduces four of the more commonly used functions. Each function is designated by an initial character of "$" followed by the unique name of the function. The name is followed by parentheses containing several MUMPS expressions (as reviewed in Section 4.1). The $LENGTH function determines the number of characters in an expression (Section 4.2). $FIND gives information about the location of one string of characters inside another (Section 4.3). $EXTRACT provides the characters at a particular position iniside a specified expression (Section 4.4). $PIECE gives specified substrings of an expression (Section 4.5). $LENGTH can also be used to count the number of pieces in a string (Section 4.6). These four functions are summarized in Section 4.7.

4.1. EXPRESSIONS

Expressions in MUMPS are composed of simple units known as "expression atoms". A common form of an expression atom is a variable name such as:

```
X
```

Another common form of an expression atom is a function like $LENGTH(NAME). An expression atom may also be a string literal, which is a sequence of characters starting and ending with a quotation mark like "TAXES". It may also be any form of number like 32.

Expression atoms are the simplest forms of expressions. Two expression atoms may be combined into a less simple expression by putting an operator between them. An example of such an expression is:

```
XRAY+3
```

which is the value of the variable XRAY plus the constant 3, or:

```
A*B
```

which is the value of A times the value of B. Other expressions may be created by connecting more atoms. For example:

```
A*B+(C*D)+E
```

The other operators that are introduced later may also be used to link expression atoms and thereby form complex expressions.

Some functions require expressions at certain positions within their parentheses to be integers. If such an expression has a non-numeric value, MUMPS first converts the value of the expression to a numeric value. (For example, the numeric value of "93" is 93; the numeric value of "-3.2BOX" is -3.2; the numeric value of "ARCH" is 0.) If the resulting numeric value is not an integer, it is truncated by removing the decimal point and any subsequent digits. (For example, the integer value of -3.2 is -3 and the integer value of 47.992 is 47.)

4.2. $LENGTH

The $LENGTH function returns the number of characters in an expression. An example is:

```
WRITE $LENGTH("ABC")
```

This causes:

```
3
```

to be written out, because three characters are enclosed by the quotation marks at the start and end of "ABC". Note that $LENGTH("") has the value 0 (zero).

It is rare to have a string literal (as in the above example) as the expression that $LENGTH manipulates. Much more commonly, the expression is the name of a variable. An example is:

```
SET L=$LENGTH(NAME),LA3=$LENGTH(ART)
```

Here, if NAME has the value "CATHERINE" then L is assigned the value of 9, whereas if NAME has the value "CATHERINE THE GREAT" then L is assigned the value of 19 with each space counting as one character.

Question 4.1. What value is assigned to LA3 in the above line if ART has the value "STUDENT"? What if ART has the value "2*3"?

A variable may be a numeric literal instead of a string literal. For example, values for NAME could be 9 or 4.7. These

produce values of $LENGTH(NAME) of 1 and 3 respectively. If NAME has the value 2*3 (as opposed to "2*3") the value of $LENGTH(NAME) is 1. Try it!

Question 4.2. Write a line of code that asks the user for his age and assigns what he enters to a variable named AGE. If his reply is only one character long, a message is displayed that says "YOU ARE DOING WELL FOR SUCH A YOUNG ONE".

4.3. $FIND

$FIND gives information about the location of one string of characters inside another. This function must be followed by two expressions in parentheses. It produces information about where the second expression ends in the first one. An example with both expressions being string literals is:

```
WRITE $FIND("ABC","D")
```

Because the first expression does not contain the second, the result returned here is:

```
0
```

An example where the second expression does occur in the first is:

```
WRITE $FIND("ABC","AB")
```

Standard MUMPS specifies that when the second expression is found in the first, the result of $FIND is a number that is one more than the ending position of the second string in the first. The second expression is "AB". These characters end at the second position in the first string. The position value of two is incremented by 1 to give the value of $FIND and so:

```
3
```

is displayed.

For $FIND, as for $LENGTH, it is more common for variable names to be used than string literals. For example:

```
SET X=$FIND(S,Z)
```

assigns to X a value that depends upon both S and Z. If S has the value "ABC" and X has the value "D", the value of 0 is assigned to X.

Question 4.3. What value is assigned to X if S has the value "ORANGE" and Z has the value "AN"?

Question 4.4. If X has the value "MUMPS Users' Group", what is written out by:

```
WRITE $FIND(X,"MUMPS")
```

and by:

```
WRITE $FIND(X," ")
```

If the value of Y is "GROUP", what value is assigned to P by

```
SET P=$FIND(X,Y)
```

Usually $FIND begins searching the first expression at its first character, but sometimes the second expression occurs more than once in the first. Therefore a third expression may be needed to specify the position of the character at which the search is to start. For example:

```
WRITE $FIND(X,"A",9)
```

requests that $FIND searches for the position of the letter A in the value of the variable X, but the third expression causes the search to start at the ninth (rather than the first) position. If the value of X is:

"A WISE OLD BIRD SAT IN AN OAK"

then:

```
$FIND(X,"A")
```

has the value 2 because the letter A is found at the first position of X. However, consider $FIND(X,"A",(9). At the ninth position of X's value is the letter L in the word OLD. The next appearance of the letter A after the ninth position is at the eighteenth position (in the word SAT). Accordingly the value of:

```
$FIND(X,"A",9)
```

is:

19

(which is 18 plus 1).

Suppose that you wanted to locate spaces in the value of X. The first one would be obtained by:

```
SET S1=$FIND(X," ")-1
```

Note that 1 is subtracted from the result of $FIND because the value returned by $FIND is the position of the first character after the space looked for. If X has the above value ("A WISE OLD...") then the first space occurs at the second position and:

```
$FIND(X," ")
```

has the value 3. Subtraction of 1 provides the value of 2, which is the position of the space. The position of the second space is then obtainable from:

```
SET S2=$FIND(X," ",S1+1)-1
```

Note that 1 is added to S1 so that the search starts at the position immediately after the first space. Thus:

```
$FIND(X," ",S1+1)
```

is equivalent to:

```
$FIND(X," ",3)
```

or

```
$FIND(X," ",$FIND(X," "))
```

Here the second $FIND function is said to be "nested" inside the first. The result is 8. Subtraction of 1 from this in the argument of the above SET command results in a value of 7 being assigned to S2.

Question 4.5. If the variable LINE has the value:

```
"TO BE OR NOT TO BE.  THAT IS THE QUESTION."
```

what values are assigned to what variables by the following:

```
SET X=$FIND(LINE,"BE"),Y=$FIND(LINE,"BE",X),Z=$FIND(LINE,"TO
BE",8)
```

4.4. $EXTRACT

The $EXTRACT function allows you to select one or more characters from within an expression. For example:

```
WRITE $EXTRACT("AVCD",3)
```

causes the character in the third position of "ABCD" to be displayed, namely C. As for $LENGTH and $FIND, the expression that is manipulated by the $EXTRACT function is usually the name of a variable. Thus if the variable QUA has the value:

```
"THIS IS THE START OF THE EXPLANATION"
```

then:

```
$EXTRACT(QUA,1)
```

has the value of T, the character at the first position of QUA. The value of:

```
$EXTRACT(QUA,3)
```

is I, the character at the third position. Note that the first expression in the parentheses of the function is the one from which a character is to be extracted, and that the second expression is an integer specifying the position of the character to be extracted.

Question 4.6. If the variable ELF has the value:

```
"A1.4B.D.83.9"
```

what values are given to the three specified variables by the following instructions:

```
SET E1=$EXTRACT(ELF,2),E2=$EXTRACT(ELF,8),E3=$EXTRACT(ELF,5)
```

It is possible to extract a sequence of characters, rather than a single character, from the first expression. THis is done by adding a third expression within the parentheses. MUMPS interprets the second and third expressions together as the starting and ending positions of the desired sequence of characters. For example:

```
SET V=$EXTRACT(MOT,3,7)
```

assigns to the variable V the third to seventh characters of the value of MOT. Thus if MOT has the value:

"A LARGE PLOT OF LAND"

then V is assigned the value "LARGE". The result of:

```
WRITE $EXTRACT(MOT,14,20)
```

is:

OF LAND

Question 4.7. The variable AUTO has the value:

"FORD,MORRIS,AUSTIN,MORGAN"

What appears in response to the following instructions:

```
WRITE $EXTRACT(AUTO,13,19),$EXTRACT(AUTO,1,4)
```

If you try to extract a character from a position beyond the end of a string, there are no characters. The result of no characters is indicated by two adjacent quotation marks. Thus, for the above value of MOT, the result of:

```
$EXTRACT(MOT,99)
```

is

""

So is:

```
$EXTRACT(MOT,30)
```

and likewise:

```
$EXTRACT(MOT,30,99)
```

because there are no characters between the thirtieth and ninety-ninth positions in the value of MOT.

4.5. $PIECE

The $PIECE function provides the MUMPS programmer with another means of pulling a sequence of characters out of some

expression. While the $EXTRACT function of the previous section allows one or more characters to be found based upon their position in the expression being analyzed, the $PIECE function allows a sequence of characters to be found based on its position relative to a specified delimiting string of characters. An example clarifies this. Suppose the variable S has the value of any sentence, such as:

"SUPPOSE THE VARIABLE S HAS THE VALUE OF ANY SENTENCE."

The first word is identified by:

$PIECE(S," ",1)

and the third word is:

$PIECE(S," ",3)

The parentheses of $PIECE first contain the expression (in this case, the variable S) to be analyzed. Second is the expression that serves as the "delimiter" or boundary. In this example, one space is used. (The character(s) of this second expression are contained in quotation marks, as required by MUMPS.) Third in the parentheses is an integer specifying the "field" chosen. If that integer has the value 1 it specifies the first field, which comprises all characters up to but not including the first instance of the delimiter. Each subsequent field is defined as the characters that begin immediately after one instance of the delimiter and end immediately before the next. Note that no single field can contain any of the delimiters.

Another example is:

SET X=$PIECE(KING,"A",3)

where the value of the variable KING is analyzed and the letter A is used as the delimiter. The characters between the second and third instance of the letter form the third field, which is to be selected. Thus if KING has the value

"AFTER THE BALL WAS OVER"

then the value given to X is:

"LL W"

Question 4.8. In the following:

WRITE $PIECE(M,".",2)

what is the expression being analyzed, what is the delimiter, and what field is sought? If M has the value:

"PLEASE WRITE. I NEED DATA. YOU CAN HELP."

What appears in response to the above WRITE command?

It is possible to use any expression for the delimiter. In particular the second expression may be a variable name in which case the delimiter is the variable's value. Thus if ID has the value:

"SMITH;JOHN;19 SEP 43;LONDON"

and D has the value:

";"

then

$PIECE(ID,D,1)

is requesting the first field of ID with a semicolon used as a delimiter. That field is SMITH. Similarly,

$PIECE(ID,D,2)

is requesting the second field of ID, which occurs between the first and second semicolons. The result is "JOHN".

Question 4.9. If ACCOUNT has the value:

"JOINT*WILSON*HAROLD AND MARY*423"

and Q has the value:

"*"

what are the values of:

(a) $PIECE(ACCOUNT,Q,1)

(b) $PIECE(ACCOUNT,Q,4)

(c) $PIECE(ACCOUNT,"I",2)

(d) $PIECE(ACCOUNT,"4",2)

and

(e) $PIECE(ACCOUNT,"*H",1)

Remember for the $EXTRACT function, that while a single character is extracted if only two expressions are specified, the addition of a third expression between the parentheses of the $EXTRACT function allows extraction of the several characters from the position specified by the second expression to the position specified by the third one. Thus:

$EXTRACT(A,B)

retrieves from the value of variable A the character at position number B, while:

$EXTRACT(A,B,C)

retrieves from the value of the variable A those characters at position numbers B to C inclusive.

In a similar way, another expression may be added inside the parentheses of the $PIECE function so that more than one field is obtained. Thus:

$PIECE(ID,D,2)

returns the second field of variable ID's value using the value of D as a delimiter. Addition of a fourth expression in the parentheses leads, for example, to:

$PIECE(ID,D,2,4)

which returns the second to fourth fields and the delimiters between them. With the value of D as a semicolon and the value of ID given above, the value of:

$PIECE(ID,D,2,4)

is:

"JOHN;19 SEP 43;LONDON"

Note that the semicolons that occur after the second field and before the fourth field are included in this result. If the instruction:

WRITE $PIECE(ID,D,1,2)

is given, the result that appears is:

SMITH;JOHN

with the delimiter between the first and second fields remaining.

Question 4.10. If ICE has the value:

"STAR OF INDIA/LONDON/WINDSOR/72/A93"

what are the results of:

(a) SET B=$PIECE(ICE,"/",4)

(b) WRITE $EXTRACT(ICE,14,16),"--",$PIECE(ICE,"/",3)

(c) SET S=$PIECE(ICE,"/",2,3)

(d) WRITE $PIECE(ICE,"OF",1),"--",$EXTRACT($PIECE(ICE,"/",5),1,2)

4.6 $LENGTH - Piece Counting

The $LENGTH function can also be used to count the number of pieces in a string as they would be found by the $PIECE function. In this case, a second expression is used in the $LENGTH function that is the delimiter by which the piecing is to be done. Using the same example, suppose the variable S has the value of the sentence:

"SUPPOSE THE VARIABLE S HAS THE VALUE OF ANY SENTENCE."

then:

$LENGTH(S," ")

would be asking for the number of pieces in the string S when a " " (space) is used as a delimiter. In the above example, the results would be 10. This use of the $LENGTH can also be viewed as a way of finding the last piece of a string. As an example, assume that the variable ADDRESS has the value:

"102 HAMILTION STREET, BILLINGS, MONTANA, 44443"

then

SET PC=$LENGTH(ADDRESS,", "),ZIP=$PIECE(ADDRESS,", ",PC)

would give the variable ZIP the value "44443".

Question 4.11. If NAME has the value:

"Smith,Winston,3rd"

What is the value of:

$PIECE(NAME,",",$LENGTH(NAME,","))

Note that determining the number of pieces in a string, a null string (no characters between two adjacent delimiters) will count as one piece. Thus:

$LENGTH("123..",".")

and

$LENGTH("123.456.789",".")

are both

3

Finally, if the delimiter evaluates to a null string, then the result is always 0. Example:

SET DEL="" WRITE $LENGTH(ADDRESS,DEL)

results in:

0

4.7. SUMMARY

Four functions have been introduced. These are $LENGTH, $FIND, $EXTRACT, and $PIECE. These functions manipulate string expressions. Sometimes an integer (whole number) value must be used for an expression. Let E1 represent the first expression in the parentheses of a function, let E2 represent the second expression, and so on. If an integer value is required, let the letter E be replaced by the letter I. Then, for example, I2 represents an integer- valued expression at the second position and I3 represents an integer-valued expression at the third position in the function's parentheses.

Using this convention, the general form for the $LENGTH function may be written as:

$LENGTH(E1)

or:

$LENGTH(E1,E2)

The first and second expressions in the parentheses may take any value.

For the $FIND function, either two or three expressions may be specified. The general form where two expressions are used is:

$FIND(E1,E2)

and for three expressions it is:

$FIND(E1,E2,E3)

While the first and second expressions may take any values, the third expression requires an integer value.

The $EXTRACT function likewise uses either two or three expressions. The general forms are:

$EXTRACT(E1,E2)

and:

$EXTRACT(E1,I2,I3)

The first expression in the parentheses may take any value but the second takes an integer value as does the third if it is specified.

The $PIECE function takes either three or four expressions as its parameters in its parentheses. The general forms are:

$PIECE(E1,E2,E3)

and:

$PIECE(E1,E2,I3,I4)

The first and second expressions in the parentheses may take any value. The third expression takes an integer value, as does the fourth expression if it is present.

CHAPTER 5

DATA

5.0. INTRODUCTION

A variable, as introduced in Section 1.4, has a name and a value assigned to it. Just as code may be stored either in the user's local workspace (Chapter 2) or on long-term storage (Chapter 3), a variable may be either a local variable and so stored locally or else a "global" variable stored on a disc or an analagous device. The similarities and differences between local and global variables are discussed in Section 5.1. Each type of variable may be given subscripts so that the same variable name may be used for groups of related data, as shown in Section 5.2. Subscripted data are associated by having the same variable name. Such associated data form a group called an "array". The concatenation operator "_" allows data to be joined that are to be stored together or otherwise manipulated as a unit (Section 5.3). The $DATA function can be used to determine whether a variable (subscripted or unsubscripted) is defined (Section 5.4). The $ORDER function allows determination of the next greater subscript used that exceeds a specified subscript value (Section 5.5).

5.1. LOCAL AND GLOBAL VARIABLES

Variables, as first noted in Section 1.4, have names and values. An example of a variable name is AGE. The value of a variable could be any expression, particularly a string of characters enclosed by quotes:

```
"THIRTY THREE"
```

It could also be a number:

```
33
```

Variables such as AGE, NAME, %ME, and so on are called "local" variables because they are specific or local to the work space where they are first defined. When two or more users are active on the same system, each may have a local variable named, say, AGE. For each user the variable's value is assigned and changed independently of what any other user does to his local variables. When use of a work space is terminated by means of the HALT command (see Section 3.8) all local variables in that work space are eliminated automatically.

It is often desirable to store for later reference a user's answer on the result of a calculation. To permit this, another type of variable exists in MUMPS. This is called a "global" variable. Possible names for a global variable are the same as for a local variable. The fact that the variable is a global rather than a local is designated by the circumflex symbol "^", which is placed before any global-variable name. Global variables could, then, be designated by ^AGE, ^NAME, and ^%ME. A value may be assigned to a global by using the SET command. For example:

```
SET ^A=1,^BAN="BOMB",^ACE=X,ACE="LOCAL"
```

assigns a value of 1 to global variable ^A and assigns the string "BOMB" to the global variable ^BAN. It also assigns to global variable ^ACE the value of the local variable X, and the local variable ACE is given the value of the string "LOCAL".

Another reason for using global rather than local variables is because data too voluminous for storing as local variables may be stored as globals. Furthermore data stored in globals may be accessed and modified by many different users. The latter point means that a programmer must be careful about assigning values to globals and killing them. Another user's data could be destroyed by accident quite easily.

Question 5.1. How would you assign the value of the local variable JOT to the global variable ^PALE?

5.2. SUBSCRIPTS

Associated with each local variable in a particular work space is a single value. It may be desirable to group or relate a set of values under the same variable name. This can be done by placing parentheses containing one or more subscripts after the variable name. For example a person's family name could be assigned to the variable D with a single subscript of "LAST" and his given name could be assigned to the same variable with a subscript of "FIRST", thus:

```
SET D("LAST")="SMITH",D("FIRST")="FRANCIS"
```

Question 5.2 Write code to assign a value of "9 OCT 1937" to the variable D with a single subscript of "BIRTH".

The value of a subscript can be any (non-null) string of printable characters or a number. In each example above, a single subscript was used but more than one subscript is possible. The following example has three subscripts:

```
TAX(1978,"INCOME",42)
```

and the next has four:

```
PATIENT(A,B,C,D)
```

Notice that in the last example variables were used as subscripts. This is possible provided each variable has the value of a non-null string or number. A subscript may itself be a subscripted variable, as in:

```
^FAT(^PERSON("SMITH"),CITY("San Francisco"),8)
```

where the first and second subscripts of the variable FAT have subscripts themselves.

Global variables may be subscripted in the same way, with non-negative integers. Therefore valid global references are:

```
^D(1)

^TAX(1976,1,42)

^FAT(PERSON("SMITH"),CITY("San Francisco"),8)
```

as well as:

```
^FAT(^PERSON(3),CITY(47),8)
```

Each variable name followed by subscripts in parentheses is called a node. A node may be either a local node or a global one. A set of nodes with the same variable name is called an array. For example all of the nodes that have a local variable name of QUOTA belong to the local array QUOTA. All nodes that have the global variable name ^FAT belong to the global array ^FAT.

The value assigned to a node may be written out using the WRITE command:

```
WRITE X(3),^MORE(A,B,9)
```

Values may be assigned to local but not global nodes by means of the READ command. For example:

```
READ "LAST NAME: ",D("LAST")
```

The KILL command may also be used on arrays, where it has a dramatic effect. The instruction:

```
KILL X
```

causes not only the variable X to be killed but also deletes all subscripted nodes of variable X. However, it has no effect on ^X if such a global exists. A subset of an array may be deleted by specifying one or more subscripts. Thus: KILL Y(4)

has no effect on variable Y or on any nodes with a first-level subscript that differs from (being less than or greater than) 4; but it causes the deletion of every node with a first-level subscript of 4. If such nodes as Y(4), Y(4,3), Y(4,2,"A",1), and Y(4,2,"A",9) exist, they are all removed by that single instruction.

Question 5.3. Write code to delete the entire part of global ^AGE that has a first-level subscript of 21 and to remove the local variable JAR together with all its subscripted nodes.

A special short-hand method is available for specifying a global node. It relies upon the record that MUMPS keeps internally of the most recent reference to an existing global node and allows the programmer to avoid repeating the name of the global and all subscripts above the level of the lowest one last specified.

For example, suppose that the most recently referenced global node exists and is:

```
^GOLD("INGOT SIZE",7)
```

If the next reference is to be to another node of global ^GOLD with a first-level subscript of "INGOT SIZE", and with a second-level subscript of any value, the specification could be made as:

```
^GOLD("INGOT SIZE",B)
```

where B represents any possible subscript value. Alternatively, the reference could be:

```
^(B)
```

Clearly far fewer characters are needed to represent a node using this method of reference than if the full reference is made. This

briefer specification is called a "naked" reference. Be careful in using it. It can be dangerous because the programmer may make a mistake about what global node was referenced last. It is best to use this capability only when you are sure to use it correctly. If you do not wish to use the naked syntax for the moment, you may ignore the rest of this section, which presents more examples. To assign values, you could use:

```
SET ^MAN(1)="ADAM",^(2)="ABEL",^(3)="CAIN"
```

which, for example, assigns to ^MAN(3) the value "CAIN". If a naked reference:

```
WRITE ^(4)
```

is made, the value written out depends upon the last global reference. If the last global reference is:

```
^ART(3)
```

then the value of:

```
^ART(4)
```

is used by the naked reference. Note that the same global name (^ART) is used and a single subscript is used for both nodes.

Alternatively, if the last global reference is:

```
^ART("WATER COLOR",5)
```

then the naked reference of ^(4) yields the value of:

```
^ART("WATER COLOR",4)
```

In this case, the global name ^ART is again correct because it is specified in ^ART(1,5). In this instance there are two subscripts, or two levels of subscripting. The first-level subscript is "WATER COLOR" and the second-level subscript is 5. The naked syntax assumes that the subscript in the parentheses of ^(4) is to be at the lowest level of ^ART("WATER COLOR",5). Thus the subscript value of 4 is to occur at the second level. The higher-level subscripts (in this case there is only one, the first-level subscript) are to be duplicated from ^ART("WATER COLOR",5).

For the naked reference of:

```
SET Y=^(8,D)
```

if the most recent full global reference is:

^JUG(A,B,C)

the value assigned to Y is:

^JUG(A,B,8,D)

Note that the global name remains the same and that all subscript values except the one at the lowest level of the full reference are adopted. Accordingly:

SET ^A(1,2)="A",^(2,3)="B",^(4,5,6)="D"

assigns to ^A(1,2,4,5,6) the value "D".

Question 5.4. In the following global nodes, what are the first level, second-level, and third-level subscripts:

(a) ^A(4)

(b) ^A(4,"*")

(c) ^A(4,"*",3)

Question 5.5. What is the result of:

SET S=^TAN("AC-469",3,1),T=^(2)

What happens if the next instruction is:

WRITE ^(3)

5.3. CONCATENATION

The concatenation operator allows two or more values to be joined together. The symbol for concatenation is the underscore " " symbol. (On some devices that do not use all of the ASCII characters, the underscore appears as a backwards-pointing arrow.) The instruction:

SET WORD="WALK"_"ING"

assigns the value "WALKING" to the variable WORD. Alternatively:

```
SET W="WALK"_END
```

assigns to the variable W a value equal to "WALK" followed immediately by the value of the variable END. If END has the value "S", then W acquires the value "WALKS", and if END has the value "ED" then W acquires the value "WALKED".

The instruction:

```
SET ^HOLD(3)=A_"HOAX"_B_C_"FIN"
```

joins together two literal (or character) strings and the values of three local variables. It assigns the resulting value to the single-subscript node of global ^HOLD as shown.

Values assigned to different local variables may be linked together being stored at a global node, thereby often reducing the space and/or time needed to store the data. Frequently a delimiter is inserted between each pair of values to allow subsequent isolation of each value using the $PIECE function described in Section 4.5. For example, the following code reads in three values and links them with colons between adjacent fields, assigning the composite value to the variable T:

```
READ "ITEM 1 ",X(1)," ITEM 2 ",X(2)," ITEM 3 ",X(3) SET T=X(1)_
":"_X(2)_":"_X(3)
```

The value of X(2) forms the second field of T using colons as delimiters. (It is assumed that the user's answers for the values of X(1), X(2), and X(3) do not contain any colons.) The second field, which should be what was entered as X(2), may be retrieved by the code:

```
$PIECE(T,":",2)
```

The value of X(1) or X(3) may be obtained as the first or third pieces.

Question 5.6. Write code that joins together the values of S,Q,R, and MAX, inserts periods to separate them, and assigns the result to the node ^JAW("TEETH","#").

5.4. $DATA

The $DATA function takes as its argument any variable name with or without subscripts, and returns an integer that specifies whether or not the particular variable or node exists. The resulting integer also denotes whether there are any levels of

subscripting below the specified node. Note that if a node has a single subscript, then all nodes with the same first-level subscript, no matter what their second-level and other subscripts, are considered to be "below" the first node. An unsubscripted variable has zero subscripts; any node with the same variable name as an unsubscripted one, but with one or more subscripts, is "below" the unscripted variable. If one node is "below" another in this way, it is said to be a "descendant" of the other.

Consider this code:

```
KILL M WRITE $DATA(M)
```

The result is 0, which is what $DATA returns when the variable in its parentheses does not exist. The value of 0 is also informing you that no nodes of M exist with one or more subscripts.

If one or more nodes exist, the result of $DATA is 01, or 10, or 11, with the first digit signifying the presence (if 1) or absence (if 0) of one or more descendants and the second digit signifying whether the specified variable has a value (1) or has not (0). Thus, the result of:

```
KILL ^M SET ^M(1)="NOTA 4" WRITE $DATA(^M)
```

is 10 since ^M has a descendant, ^M(1), but has no value of its own.

Question 5.7. What values are assigned here:

```
KILL X SET X(1)="EGG",X(2)=$DATA(X)
```

If the variable inspected by the $DATA function has a value but no descendant array elements, the value of $DATA is 01, which in practice appears simply as 1. Thus:

```
KILL SUM SET SUM=0 WRITE $DATA(SUM)
```

results in the value:

```
1
```

appearing.

Finally, if a variable has a value and there are also subscripted nodes to which values have been assigned, the result of $DATA is 11. The first digit symbolizes the existence of one or more descendants while the second digit shows that the

specified variable has an assigned value. For example:

```
KILL QUE SET A="A",QUE=4,QUE(A,"location",3)="TAIL",X=$DATA(QUE)
```

assigns a value of 11 to the variable X.

The variable inspected by $DATA may have subscripts, as in $DATA(QUE(A)). If the above line of code has been executed, QUE and QUE(A,9,3) are defined but QUE(A) does not have a value assigned. Note that QUE(A,9,3) is considered a descendant of QUE(A) because the former is identical to the latter apart from having some additional subscripts. The value of $DATA(QUE(A)) is then 10, as is the value of $DATA(QUE(A,"location")). However the value of $DATA(QUE(A,"location",3)) is different. It is 1. This is because a value has been assigned to QUE(A,9,3) but there are no nodes of QUE with four or more subscripts led by those three subscripts of QUE(A,"location",3). The concept to grasp is that of "descendants".

Question 5.8. The following variables have values assigned: A, B, ^C, A(3), B(42,1), ^C(9), A(3,7), B(1), ^C(9,"STRING",1,4), and B(42,2). What are the descendants of:

(a) A

(b) B

(c) ^C

(d) A(3)

(e) B(42)

(f) ^C(9)

(g) ^C(9,"STRING",1)

(h) ^C(9,"STRING",1,4)

Question 5.9. With values assigned to variables as specified for Question 5.8, what are the values of:

(a) $DATA(A)

(b) $DATA(B(42,1))

(c) $DATA(^C(9))

(d) $DATA(E)

(e) $DATA(B(42))

5.5. $ORDER

The $ORDER function takes in its parentheses a subscripted variable. It finds the next larger subscript at the last level of subscripting specified. For example:

^G(4,91)

has two levels of subscripting. The value of:

$ORDER(^G(4,91))

is 92 if the node ^G(4,92) exists. The node could exist because it has a value assigned (so that $DATA(^G(4,92)) could have a value of 1), or because a node of ^G exists with a first-level subscript of 4, a second-level subscript of 92, and a third-level and perhaps lower subscripts of any value (so that $DATA(^G(4,92)) has a value of 10). Alternatively both reasons could apply (so that $DATA(G(4,92)) has a value of 11.

If the node ^G(4,92) is not defined, the node with the next larger subscript is sought for $ORDER(^G(4,91)). If no larger subscript is found, MUMPS returns a value of null for $ORDER(^G(4,91)). The null is, of course, an illegal subscript. It is used as a convention to show that no legal subscript is found.

The following examples should reinforce the above. The code:

```
KILL A SET A(1)="A",A(2)="B",A(3)="C",A(20)="T"
```

creates four nodes of the A array. The value of:

```
$ORDER(A(1))
```

is 2. The result of:

```
WRITE $ORDER (A(2))
```

is that 3 is written out. The value assigned to X by:

```
SET X=$ORDER(A(3)),Y=$ORDER(A(20))
```

is 20 and the value assigned to Y is "",null, because there is no

larger subscript than 20. Some additional results for $ORDER are that:

```
WRITE $ORDER(A(0))
```

produces 1, while

```
WRITE $ORDER(A(12))
```

causes 20 to be displayed. Note that in these two cases, the specified subscript at the lowest level in the $ORDER function does not have to exist. Thus, although the value of $DATA(A(12)) is 0, A(12) can be used in the parentheses of $ORDER. A special case of being able to use a subscript for an undefined node is that a subscript of null may be used at the lowest level of subscripting in the $ORDER function. For example:

```
WRITE $ORDER(A(""))
```

is allowed. It causes 1 to be written out. If A(1) does not exist but A(2) still does, the above instruction causes 2 to be written. Alternatively if A(0) exists, the value written is 0. The use of null within the parentheses of $ORDER is often referred to as "priming".

Suppose the nodes shown in Table 5.1 have been assigned values. Nodes with a single subscript are at the left, with an increasing number of subscripts towards the right. Below are listed instances for $ORDER on first-level nodes and the corresponding value of each instance:

```
$ORDER(P("")) has the value 1
$ORDER(P(0))      "    "    1
$ORDER(P(1))      "    "    2
$ORDER(P(2))      "    "    9
$ORDER(P(3))      "    "    9
$ORDER(P(9))      "    "   (a null value)
```

Second-level nodes with $ORDER give the following results:

```
$ORDER(P(1,"")) has the value 2
$ORDER(P(1,2))      "      "   9
$ORDER(P(1,9))      "      "  (a null value)
$ORDER(P(2,""))     "      "   0
$ORDER(P(2,0))      "      "   1
$ORDER(P(2,1))      "      "   4
$ORDER(P(2,4))      "      "  (a null value)
```

Question 5.10. What are the values of:

(a) $ORDER(P(9,""))

(b) $ORDER(P(9,0))

(c) $ORDER(P(9,1))

(d) $ORDER(P(9,7))

(e) $ORDER(P(9,8))

1 subscript	2 subscripts	3 subscripts	4 subscripts
P(1)	P(1,2)	P(1,2,3)	P(1,2,3,4)
	P(1,9)		
P(2)	P(2,0)	P(2,0,8)	
	P(2,1)		
	P(2,4)	P(2,4,1)	
		P(2,4,8)	P(2,4,8,3)
P(9)	P(9,0)	P(9,9,1)	
		P(9,9,3)	

Table 5.1. Nodes of the local array P to which values have been assigned.

(f) $ORDER(P(9,9))

(g) $ORDER(P(9,10))

Third-level nodes with $ORDER give the following results:

```
$ORDER(P(1,2,"")) has the value 3
$ORDER(P(1,2,1))      "       "   3
$ORDER(P(1,2,3))      "       "  (a null value)
$ORDER(P(2,0,""))     "       "   8
$ORDER(P(2,0,8))      "       "  (a null value)
$ORDER(P(2,4,""))     "       "   1
$ORDER(P(2,4,1))      "       "   8
$ORDER(P(2,4,8))      "       "  (a null value)
```

Question 5.11. What are the values of:

(a) $ORDER(P(9,9,""))

(b) $ORDER(P(9,9,0))

(c) $ORDER(P(9,9,1))

(d) $ORDER(P(9,9,2))

(e) $ORDER(P(9,9,3))

(f) $ORDER(P(9,9,4))

Fourth-level nodes with $ORDER give the following results:

```
$ORDER(P(1,2,3,"")) has the value 4
$ORDER(P(1,2,3,0))      "       "   4
$ORDER(P(1,2,3,4))      "       "  (a null value)
$ORDER(P(2,4,8,""))     "       "   3
$ORDER(P(2,4,8,3))      "       "  (a null value)
```

If a value is assigned to the node P(7,3), the value of $ORDER(P(2)) changes. As shown earlier, its value with the nodes in Table 5.1 is 9. With the new node its value is 7, even though no value has been assigned to P(7). In effect, $ORDER returns the next higher subscript value for which $DATA is nonzero. (Remember from the previous section that $DATA(P(7)) would be 10 after P(7,3) is assigned a value but 0 before that. Also $DATA(P(9)) in Table 5.1 is 11 and $DATA(P(1,9)) is 1.)

Question 5.12. After the following instructions:

```
KILL R SET R(8)="S",R(90,1)="C",R(7,3)="A",R(3)=2,R(90,R(3))=7
```

what are the values of:

(a) R(90,2)

(b) $ORDER(R(""))

(c) $ORDER(R(2))

(d) $ORDER(R(4))

(e) $ORDER(R(7))

(f) $ORDER(R(2))

(g) $ORDER(R(7,1))

(h) $ORDER(R(90,1))

(i) $ORDER(R(90))

(j) $ORDER(R(8))

Up until now, we have talked about the $ORDER function only with numeric subscripts. Of course, any string can be used as a subscript, not just numbers, and the $ORDER will also work with any string subscript as well.

When we talked about numeric subscripts, $ORDER always gave us the next "higher" subscript. Thus, numbers are always presented in numeric order. With strings, $ORDER will give the next higher subscript in alphabetical order. Example:

```
SET T("A")=1

SET T("C")=2
```

then:

$O(T("AAB")) has the value of "C"

and:

```
SET BC1("ABCD")=3

SET BC1("ABCF")=4
```

then:

```
$ORDER(BC1("")) has the value of "ABCD"
$ORDER(BC1("A"))     "      "     "ABCD"
$ORDER(BC1("ABCD")   "      "     "ABCF"
$ORDER(BC1("ABCF")   "      "     (a null value)
```

The order in which subscripts are presented is called the collating sequence. $ORDER collates subscripts according the sequence listed in Table 9.1. In general, all numbers will be collated first, followed by all uppercase letters, then all lower case letters. $ORDER and string subscripts will be discused in more detail in chapter 9.

CHAPTER 6

MORE COMMANDS

6.0. INTRODUCTION

Section 6.1 reviews the four commands for manipulating data (namely WRITE, READ, SET, and KILL) that were introduced in Chapter 1. Section 6.2 reviews the four commands (IF, GOTO, DO, and QUIT) from Chapter 2.

Three other frequently used commands are ELSE, FOR, and HANG. The ELSE command is the complement of the IF command, as shown in Section 6.3. The FOR command allows the repeated execution of all the commands that follow it on a line (Section 6.4). The HANG command specifies a length of time for which a pause is to last (Section 6.5). A similar feature is the "timeout", which is used to specify a maximum time during which MUMPS will await the completion of certain commands (such as the READ command) as shown in Section 6.6.

6.1. REVIEW OF COMMANDS FOR MANIPULATING DATA

The WRITE command introduced in Section 1.2 may be used on global variables as well as local variables, strings of characters, and numbers. For example if the global node ^HO(3,4) has the value "CAROL", then the instruction:

```
WRITE ^HO(3,4)
```

results in the display of:

```
CAROL
```

If the local node ART("ARCHIVE") has the value "N3P4", then the instruction:

```
WRITE ART("ARCHIVE")
```

results in the display of:

```
N3P4
```

The READ command (see Section 1.5) may be used to assign a value to any local variable with or without subscripts. For example:

```
READ "NAME: ",N," AGE: ",N("AGE")," NATIONALITY: ",N("HOME",4)
```

prompts the user to enter his name, which is assigned to the variables N, his age (assigned to the node N("AGE")), and his nationality, which is assigned to the node N("HOME",4). The READ command may be used to assign a value to any local variable, but not to a global variable.

A value may be assigned to either a local or a global variable by means of the SET command, as introduced in Section 1.6. For example:

```
SET A=1.3,B("EXI")="DAYTON",^AX(A,4)=B("EXI")
```

assigns the value of 1.3 to the local variable A as well as the string "DAYTON" to the singly subscripted node B("EXI"). In addition, a node of global ^AX is created, with a first-level subscript equal to the value of local variable A (which is 1.3) and a second-level subscript of 4. To this node is assigned the value of the local node B("EXI").

The fourth command for manipulating data is the KILL command (see Section 1.7). KILL may be used on both local and global variables. As noted in Section 5.2, when the KILL command is used on a variable that has subscripted nodes, it can eliminate more nodes than those specified. This is because the KILL command removes any nodes that are descendants of the variables specified. To illustrate this further, consider the local array P shown in Table 5.1 If the instruction:

```
KILL P(9)
```

is given, not only P(9) disappears but so do P(9,0), P(9,9), P(9,9,"S"), and P(9,9,"JI"). Likewise the instruction

```
KILL P(1,2,3)
```

removes not only P(1,2,3) but also P(1,2,3,4) while:

```
KILL P
```

removes all of the nodes shown in Table 5.1.

6.2. REVIEW OF FOUR MORE COMMANDS

The other four commands introduced previously are IF (Section 2.2), GOTO (Section 2.6), QUIT (Section 2.7), and DO (Section 2.8). The IF command is needed in order to determine which of two actions should be taken. These two actions are either to execute the code on the line following the IF command or to ignore that code and proceed to the line immediately below the one containing the IF command. For example, the code:

```
IF A="JONES" SET B=4 WRITE $PIECE(S,".",3)
```

causes the value of the variable A to be tested for equality to "JONES". If the value of A is "JONES", the argument of the IF command is true and the SET and WRITE commands are executed. Otherwise those commands are bypassed.

In a sense, the IF command can cause execution of code to branch away from one point to another. The GOTO and DO commands also cause a change in the place where the code is being executed. With the GOTO command, the transfer is permanent. Thus:

```
IF $LENGTH(ANSWER)>40 GOTO TOOLONG
```

inspects the length of the variable ANSWER and causes branching to the line labelled TOOLONG if there are more than 40 characters in the value of ANSWER. Similarly:

```
IF $EXTRACT(X,1)="A" GOTO ^ALPHA
```

inspects the initial character of variable X's value for equality to the letter "A" and causes branching to the start of the routine named ALPHA if an equality is found. It would be pointless to put any commands after a GOTO command on the same line because they could not be executed.

Alternatively, while the DO command causes branching like the GOTO command, the additional effect of the DO command is that when the lines of code invoked by the DO command are finished, any code following that DO command is executed. THUS:

```
IF N<1 DO LINE^CORE SET N=1
```

inspects the value of the variable N and branches to the line labelled LINE of the routine named CORE; when no further code exists to be executed in routine CORE, or a QUIT command is encountered, the SET command after the DO is executed.

The fourth command in this group is QUIT, which causes execution of code to end. QUIT is particularly important for ending code invoked by the DO command so that execution of code returns to the next command after the DO.

6.3. THE ELSE COMMAND

The first new command of this Chapter is ELSE. It does not take an explicit argument. Its effect depends upon whether the most recently executed IF command was true or false and is opposite to the effect of the IF command. For example:

```
IF AGE>18 SET TAX="Y"
ELSE  SET TAX="N",B=B-1
```

begins by checking the value of AGE. Note that ELSE takes no argument and so it is followed by two spaces before the next command. When it is true that AGE exceeds 18, the value of "Y" is assigned to TAX; the ELSE command effectively has a false-valued argument because the argument of the previous IF command is true; in such a case, the remaining commands on the same line are not executed. Alternatively, if the value of AGE does not exceed 18, then the argument of the IF command is false; the ELSE command effectively has a true-valued argument; the commands following ELSE are executed, assigning a value of "N" to TAX and reducing the value of B by 1. Observe that the ELSE command appears in a different line from (and often the next line later) an IF command to which it is related.

Question 6.1. If NAME has the value "SMITH", AGE has the value 99, YEAR has the value 1945, and S has the value "A1", which of the following expressions are true:

(a) NAME=99

(b) YEAR=1945

(c) S="A1"

Question 6.2. With the same values as in Question 6.1, what is the result of the following code:

```
IF YEAR<1978 SET P="A"
ELSE  WRITE "CONTINUING"
```

and what is the result of:

```
IF S="B1" WRITE "B" DO SUB
ELSE  READ "NEW VALUE: ",S
```

Question 6.3. For the second example in Question 6.2, what effect on the result of the ELSE command could an IF command in line SUB have? (Note, be aware of such danger in using the ELSE command.)

6.4. THE FOR COMMAND

The FOR command is used to achieve repeated execution of all commands following FOR on a line. To achieve this, the argument of the FOR command consists of the assignment of different values to a local variable; subsequent commands are executed for each assigned value. The following code:

```
FOR I=1,9,8 SET A(I)=I DO SUB
```

assigns successively values of 1, 9, and 8 to the local variable I. For each value of I, a node of the A array is given that value and the subroutine beginning at line SUB is executed. Note that the different values assigned to I are separated by commas.

Another example is:

```
FOR J=1,2,3,4,5,6 READ " VALUE: ",X(J)
```

where a sequence of six values is assigned to J and the user is asked for six responses that are assigned to different nodes of the X array. In this example the values assigned in the argument of the FOR command differ from each other by one. As it is quite common for the values to change in constant steps, MUMPS contains a special syntax to designate this: the initial numeric value is specified, followed by a colon and then the difference between successive values; another colon appears followed by the maximum value wanted. Accordingly, the previous line of code could be abbreviated to:

```
FOR J=1:1:6 READ "VALUE: ",X(J)
```

This code reads "for values of J starting at 1 and incremented by 1 up to a value of 6, read...". The commands following the FOR command are executed only as long as the value of J does not pass the value shown after the second colon.

Question 6.4. What values does K take in the following:

(a) `FOR K=3:2:13 DO A`

(b) `FOR K=4:-1:0 DO B`

(c) `FOR K=1,2,8:1:14 DO C`

The commands that may be executed for a particular value for a FOR-loop variable are said to be within the "scope" of the FOR command. A QUIT command can be used in the scope of the FOR command, to terminate execution of the FOR command and its scope. For example:

```
FOR F=1:1:6 READ " VALUE: ",X(J) IF X(J)="" QUIT
```

requests between one and six values to be entered by the user. If the user enters no characters but merely presses the entry key for any prompt, the corresponding value of X(J) is the empty string so that the expression:

```
X(J)=""
```

is true and the QUIT command is executed. This ends the FOR loop: J is incremented no further and no more values are requested from the user. If the user always enters a value, the FOR loop does not end until the maximum value of the FOR loop has been used.

Alternatively, it may be preferable to finish executing the FOR command only when a QUIT command is encountered. In this case, no maximum value is specified in the argument of the FOR command. An example is:

```
FOR Y=1960:1 DO ^SUM IF TOTAL>1000 QUIT
```

where the value of variable Y is 1960 initially and the routine named SUM is executed; on completion of that routine, the value of the variable TOTAL is tested. If TOTAL exceeds 1000, no further use is made of the FOR command. Otherwise, the value of Y is increased by 1 and the DO and IF commands are again executed. This continues until the value of TOTAL exceeds 1000. Note that the FOR variable remains defined after the FOR loop terminates, with the value it attains in the last iteration of the loop.

Question 6.5. WHen does execution end for the following lines of code:

(a) `FOR I=1,4,9 SET X(I)="A"`

(b) `FOR J=8:-2:-8 SET M=M+J DO SUB`

(c) `FOR K=4:3 WRITE K`

(d) `FOR K=4:3  WRITE K IF K>20 QUIT`

(e) `FOR L=1,9,4:1:8,2 DO LOOP IF L=7 GOTO FRANCE`

6.5. THE HANG COMMAND

The HANG command allows the programmer to specify a time for which activity pauses. FOr example:

```
HANG 8
```

causes execution to pause for 8 seconds. Similarly:

```
WRITE "WAITING " HANG 10 WRITE "STILL WAITING"
```

causes WAITING to be displayed; then there is a 10-second pause before the next WRITE command is obeyed. A variable may be the argument of the HANG command, as in:

```
HANG TIME
```

where the value of TIME specifies the number of seconds paused. If the value of TIME is 3, then the pause lasts for 3 seconds. If the value of TIME is:

```
"AFTER A WHILE"
```

the numeric value of that phrase is 0 (zero) and so the pause lasts 0 seconds.

HANG is particularly useful after some text has been displayed, so that the text remains on the screen while the user reads it. For example, after LI lines of text have been displayed, the instruction could be:

```
HANG LI*3
```

which causes a pause of 3 seconds per line. Note that if HANG is abbreviated to its initial letter and used without an argument, it is interpreted as the HALT command.

Question 6.6. Write code that requests the user to enter a value for his age, assigned to variable YEARS. If that value is over 65, pause for 10 seconds before branching to line OLD.

6.6. THE TIMEOUT

The "timeout" feature provides a different way for MUMPS to pause before continuing. It allows a maximum time to be specified during which MUMPS waits for successful completion. A timeout is used most commonly for the READ command and is specified by placing a colon after the argument of interest followed by the number of seconds that specifies the maximum period to wait. For example:

```
READ "NAME ",N:60
```

specifies that the user is to be asked for his name but that if he does not answer within 60 seconds, MUMPS is to stop waiting for

his response and move to the next instruction. For the READ command, if the user makes no reply within the specified time, the value of the empty string is assigned to the variable (which is N in the above example). However, if part of the answer is typed in, that part entered before the time runs out is assigned to the variable designated.

The period to be waited may be specified by a variable rather than a numeric constant. This is the case in:

```
READ "CONTINUE? ",X:T
```

where the computer is to wait for no more than T seconds for the user to reply. If the value of T is 45, then 45 seconds are allowed for the user to make his specification. If the value of T is "5A", the numeric value of T is 5 seconds and so the system allows the user only 5 seconds in which to respond.

Question 6.7. Write a line of code that pauses 10 seconds, asks a user to reply to a quesion "INVENTORY OPTION? ", with a timeout of 1 minute, assigning his response to the variable OPT.

Use of a timeout causes a side effect that is relevant to the IF (Section 2.2) and ELSE (Section 6.3) commands. MUMPS makes use of a feature called the "Test Switch" whose value is affected by an IF command and also by a timeout. When the argument of an IF command is true, having a truth value of 1, the Test Switch is given a value of 1; when the argument of an IF command is false, its truth value is 0 and the Test Switch is given a value of 0. For a timeout, the value assigned to the Test Switch depends upon whether the desired action is performed within the specified time. Specifically for a timeout on an argument of a READ command, if an answer is entered before the time expires, the Test Switch is given a value of 1; otherwise it is given a value of 0. The value of the Test Switch may be inspected by means of IF or ELSE without an argument (and therefore with two spaces before the next command). For example:

```
READ "NAME ",N:100 IF  SET ^ID(A)=N
```

stores the value of the name only when an answer is entered within 100 seconds.

```
READ "HOW ARE YOU? ",ANS:600 ELSE  DO WAKEUP
```

performs the WAKEUP subroutine when an answer is not entered within 10 minutes.

CHAPTER 7

STYLE IN USER INTERACTION

7.0 INTRODUCTION

In order to help the user of a MUMPS application, the messages presented by the application should be clear and consistent (Section 7.1). Interaction by the user can be facilitated by judicious formatting of data using new lines and tabbing capabilities (Section 7.2.). The $JUSTIFY function provides an additional formatting tool (Section 7.3).

7.1. PROMPTS AND EXPLANATIONS

A prompt is the text displayed to a user to indicate that a response should be entered. For example:

```
READ "ANSWER: ",ANS
```

displays the prompt:

```
ANSWER:
```

Likewise:

```
READ "NAME: ",N," ROOM: ",R
```

prompts first with the text:

```
NAME:
```

and then after one answer has been entered (which is assigned o variable N) the second prompt appears:

```
ROOM:
```

to which the user replies with a second answer. Sometimes the prompt might be quite long, as in:

```
READ "CHOOSE FROM THE AVAILABLE ROOMS. ENTER S FOR SINGLE, D FOR DOUBLE, N FOR NORMAL MEETING ROOM, L FOR LARGE MEETING ROOM: ",R
```

Here the user is being asked for an answer that is one of four possibilities. If the prompt is simplified as in the following:

```
READ "ROOM: ",R
```

the user may be unsure how to reply. A more useful simplification

of the long prompt could be into a list of possible answers without the explanatory details:

```
READ "S,D,N, or L: ",R
```

or

```
READ "S/D/N/L: ",R
```

Such prompts are valuable to speed user interaction, by minimizing the number of characters that have to be written on the terminal and the time needed for the user to read and comprehend what appears as the prompt.

User interaction is facilitated if the prompts are what the user expects or wants. In particular, the user that is familiar with an application wants brief prompts where as he wants longer prompts if he is new to it or has not used it for a while. When several prompts are to have the form of lists of possible answers, it is best if they have similar appearances throughout an application. For example, they should always appear separated by commas:

```
READ "S, D, N, or L: ",R
```

or always by slashes

```
READ "ROOM (S/D/N/L): ",R
```

If alphabetic characters are used for some of the characters allowed as answers, such characters should be used in all answers. Likewise, if the programmer prefers to get the user to enter numbers, he should require that numbers be entered in all cases possible. In this way, the surprises experienced by the user are minimized and the interaction can proceed rapidly.

It is helpful for the user to be able to obtain an explanation. For example, an explanation could be given in response to a question mark, as in the line below that starts with the label ROOM:

```
ROOM READ !,"S/D/N/L: ",R IF R["?" WRITE " ENTER S FOR SINGLE
ROOM, D (DOUBLE), N (NORMAL MEETING ROOM), or L (LARGE MEETING
ROOM)" GOTO ROOM
```

When a question mark is entered, a longer message is given to designate more clearly what is wanted. Then the GOTO command is executed, to return execution of code to the start of the line labelled ROOM.

Question 7.1. Write a line of code labelled ITEM, which prompts briefly for the user to select one of eight possible

items. Check whether the user enters a question mark. If he does, give him an explanation of what is wanted and branch back to the start of the line.

Instead of inspecting the user's answer for a question mark, the programmer could check whether it failed to contain one of the desired answers. For example, the value of R could be tested by:

```
IF R'="S",R'="D",R'="N",R'="L" WRITE ...
```

where ... indicates the longer message. Remember from Section 2.4. that the apostrophe (') symbolizes negation. Thus the argument of the IF command says that if the value of R is not the letter S nor the letter D nor N nor L, then the subsequent (WRITE) command is to be executed. An alternate way of making the same test is to check that the value for R entered by the user is only one character long (by means of the $LENGTH function) and then to use the $FIND function thus:

```
IF $LENGTH(R)=1,$FIND("SDNL",R) GOTO GOOD
```

An acceptable answer would cause the GOTO command to be executed. Otherwise execution would drop to the next line, on which should be a WRITE command followed by the error message and then a GOTO command branching the user back to the prompt.

Question 7.2. Write code beginning at the line labelled NET in which a value for a television network is requested. That value is assigned to the local variable X. If it does not have the value A (for ABC), C (for CBS), N (for NBC), or P (for PBS), inform the user of that and request that the correct value be entered.

7.2. NEW LINE, NEW PAGE, AND TABBING

Just as user interaction can be facilitated by prompts that are in a style consistent with each other, so are transactions smoothed by sensible placement of prompts, answers, and messages. In particular, it is wise to begin most prompts and error messages on a new line. This is done by means of the "carriage-return/line-feed" formatting operator that is symbolized by the exclamation mark "!". It can appear as an argument of a WRITE or a READ command. It specifies that the carriage or cursor on the terminal is to be returned to the start of the line and then fed down one line to start a new one. For example:

```
WRITE 1,2,3
```

causes:

```
123
```

to appear while:

```
WRITE 1,!,2,!,3
```

causes:

```
1
2
3
```

to appear. Similarly:

```
WRITE "A",!,"B",!,"C"
```

causes the display of:

```
A
B
C
```

Likewise: `FOR J=1:1:5 WRITE !,J,": " READ X(J)`

causes the printing of numbers 1 to 5 followed by a colon at the start of successive lines; following each colon, the computer waits for the user to enter a value that is assigned to a node of the X array. The transaction might appear as follows (with the user's answers underlined to distinguish them from what the computer displays):

```
1: SMITH
2: FENDERSON
3: APPLEMAN
4: ROYSTON
5: GRACEFIELD
```

Note how each number is written on a new line and that each user's answer begins in the same column. The value "SMITH" is assigned to X(1), "FENDERSON" is assigned to X(2), and so on. The FOR command (introduced in Section 6.4) is useful here, allowing a single line of code to perform the actions that would otherwise require several.

Question 7.3. What MUMPS code would display a prompt for the user's name on one line, assign his answer to the variable NAME, and then ask him for his age at the start of the next line,

assigning that response to the variable AGE?

Question 7.4. Global nodes ^TEXT(3,1), ^TEXT(3,2), and so on to ^TEXT(3,8) contain text to serve as prompts. What code would display those eight prompts, each on a new line, and obtain from the user a response that is assigned to the nodes ANS(1), ANS(2), and so on to ANS(8)?

If it is desirable to begin input or output at the top of a new page (as opposed to the start of a new line), the new-page or form-feed operator is used. That is denoted by the pound sign, "#". For example:

```
WRITE !,"NEW LINE",#,"NEW PAGE"
```

Tabbing to a specified column is possible in MUMPS by means of the question mark "?" followed by one or more digits. This construction, like the carriage-return/line-feed, may appear as an argument of any WRITE or READ command. For example:

```
WRITE !,?4,A
```

causes a new line to be started, four spaces to be passed over, and then the value of A to be written. The value of A starts in the fourth column, which begins after four spaces from the left. Note that by definition in MUMPS the first or left-most column is column 0 and the second column is called column 1. That is reached by:

```
WRITE !,?1
```

The values of variables AX, BX, and CX can be written starting respectively in columns 0, 30, and 60 by means of:

```
WRITE !,AX,?30,BX,?60,CX
```

If the length of AX's value is more than 30, then the cursor or carriage is beyond the thirtieth column when the first tabbing argument (?30) is reached: the cursor/carriage is not moved backwards to the specified column but remains where it is so that there is no space between the end of AX's value and the display of BX's.

Question 7.5. (a) For the following code:

```
WRITE !,"NAME",?7 READ NAME
```

in what column does the user start writing his value for the variable NAME? (b) How many spaces are there between the end of the prompt "NAME" and the start of the user's reply?

A variable or other expression may be used instead of a numeric constant after the tabbing symbol. For example:

```
WRITE !,?SPACE,X
```

starts a new line and tabs over so that the value of X starts in column number SPACE.

Question 7.6. Write code that displays five asterisks on five lines to form a chevron (>), the first and fifth asterisks not being indented, the second and fourth appearing in column two, and the third appearing in column four.

7.3. $JUSTIFY

The formatting operator (?) for tabbing is useful for presenting a series of data starting in the same, selected column of different lines. The $JUSTIFY function also provides a formatting capability, displaying data that are aligned on the right instead of on the left. Data displayed thus are said to be "right justified". An example is:

```
FOR J=1:1:8 WRITE !,?4,YAR(J)
```

which causes the display of eight values starting at column four on eight successive lines. However:

```
FOR J=1:1:8 WRITE !,$JUSTIFY(YAR(J),10)
```

causes the display of these same eight values of YAR(J) in eight successive lines, ending (rather than starting) in the same column. Two expressions appear as the parameters of $JUSTIFY. The first expression specifies what is to be written out. In the example, it is the value of YAR(J). The second expression in the parentheses specifies the number of character positions in which the value is to be written. This is often called the "field

width". In the above example, the field width is 10. Accordingly the first 10 columns are used for the display. Those columns are column 0 (the first column) to column 9. Because of the $JUSTIFY function, each value of YAR(J) ends at column 9 on its line. For example, if YAR(1) has the value "*", YAR(2) has the value "ABCDEFGHIJ", and YAR(3) has the value "TEST", the first three lines would appear as:

```
         *
ABCDEFGHIJ
      TEST
```

For a value longer than the specified field width, the display merely overflows the field. So, if the value of YAR(2) is changed to "ABCDEFGHIJKL" then it is two characters longer than the 10 characters expected. The first lines become:

```
         *
ABCDEFGHIJKL
      TEST
```

Question 7.7. Write code that displays 1 to 15, each on a new line with the numbers aligned to the right. Each number is to be followed by a period and a space. Then the user is to enter a value that is assigned to the global variable ^CARE(4,N), where N is value of the displayed number.

When the $JUSTIFY function is used with WRITE, more than one such function maybe used for a single line. This is possible because the field width is counted from where the last character was written; that character does not have to be the carriage-return/line-feed. For example:

```
WRITE !,?6,"TYPE",$JUSTIFY(V,9),$JUSTIFY(W,4)
```

causes six spaces to appear at the start of a new line, thus occupying column 0 to column 5. The four characters TYPE are displayed next, occupying column 6 to column 9. The value of V is written in a field width of nine characters (as specified by the second expression in $JUSTIFY). It thus appears in columns 10 to 18. Finally the value of W is written right-justified in the next four columns, which are column 19 to column 22.

Question 7.8. What code would display the value of variable S on a new line, followed by the value of T right-justified in a field of width 10 characters, and ended by the value of R starting in column 40?

$JUSTIFY may be used with three expressions instead of two. The first expression should be a numeric expression; the second expression has a numeric value for the field width; the third expression has a numeric value that designates the number of digits to which the first expression is to be rounded after its decimal point.

It is most common to use $JUSTIFY in the argument of a WRITE command. Occasionally it appears with the READ and SET commands and in other contexts.

CHAPTER 8

ANSWER CHECKING

8.0. INTRODUCTION

The user of an application may enter an unexpected (and therefore not necessarily correct) response to any question. Every answer should therefore be checked for correctness. The "contains" operator provides a concise method of checking whether certain characters are in a user's answer (Section 8.1). The "patter-match" operator is a powerful means of testing an answer for the types of character it contains (Section 8.2). Such tests may be made even more powerful by combining them through the logical "and" and "or" operators (Section 8.3).

8.1. THE "CONTAINS" OPERATOR

In Section 4.3, it was shown how the $FIND function locates where one sequence of characters ends within another. For example:

```
IF $FIND(NAME,Z)>0 DO YES
```

branches to the line labelled YES if the value of Z is found in the value of NAME> THat could be abbreviated to:

```
IF $FIND(NAME,Z) DO YES
```

because if the resulting value of $FIND is 0, that represents a false truth value, it being false that the value of Z is found in the value of NAME. Otherwise Z does occur in NAME and the value of $FIND is a number of 1 or more, which in MUMPS has a value of true. (See Section 2.4 for more discussion of truth values.)

It may be unnecessary to locate where one expression value occurs in another, but only whether or not it occurs. Then the contains operator, represented by the opening square bracket "[", may be used rather than $FIND. The above $FIND example may be rewritten with code that uses the contains operator:

```
IF NAME[Z DO YES
```

Note that fewer characters are needed to use the contains operator instead of the $FIND function. Some other examples are:

```
IF ANSWER["?" DO EXPLAIN3 GOTO REP
```

which checks whether the value of ANSWER contains a question mark and:

```
IF PLANE["BOEING" SET TYPE=4
```

which checks whether the value of PLANE contains a particular sequence of characters.

Question 8.1. Which of the following are true if M has the value of "A43", N has the value "BART", P has the value "10:30 PM", and CH has the value "A":

(a) `"XYZ"["X"`

(b) `"XYZ"[CH`

(c) `M[CH`

(d) `N[CH`

(e) `P[CH`

(f) `CH[M`

Question 8.2. Write code to test whether the variable MEDS1 contains the value "DIAXZEPAM". If it does, increase the value of SUM by 1. What change would you make to the code if the test was to be for MEDS1 having the precise value of "DIAZEPAM"?

Question 8.3. Write code to determine whether the variable ANSWER contains the character "Y".

8.2. THE "PATTERN-MATCH" OPERATOR

The pattern-match operator is represented by the question mark (?). Do not worry that the question mark also denotes the tabbing operator for formatting data (as seen in Section 7.2). There is no danger that MUMPS will misunderstand what you mean when you use the question mark. This is because your meaning will be defined uniquely by the context in which the question mark appears; typically tabbing occurs in conjunction with READ and WRITE while pattern-matching occurs in conjunction with the IF command.

The syntax for the pattern-match operator requires the specification of the expression (often the name of the variable) being checked, followed by the question mark. Then one of more patterns may be specified. Each pattern contains either:

-- One or more characters enclosed in quotes such as "FUN", which causes the three characters FUN to be sought in the variable being checked; or

-- A particular alphabetic character that designates a group or class of character.

The particular characters used to specify patterns are:

A, for any alphabetics.
E, for any characters at all.
L, for lower-case alphabetics.
N, for numeric digits.
P, for punctuation characters.
U, for upper-case alphabetics.

Also, allowed but rarely used is the character C, which categorizes the 33 "control" or "nonprinting" characters. These control characters are all characters remaining after the 52 alphabetics (upper and lower case), 10 numerics, and 33 punctuation characters are removed from consideration.

Question 8.4. What does a pattern containing the character A check for? What change would occur in what was checked for if surrounding quotation marks were added so that "A" appeared in the pattern?

The specification of the particular characters or character type must be preceded by a number denoting how many times that sequence of characters or that type of character is to appear. This completes the specification of a pattern. Some examples of patterns are:

3A

which check for the presence of three alphabetic characters;

2"*"

or:

```
1"**"
```

both of which look for the presence of two adjacent asterisks; and

```
4N3A
```

which checks for the presence of four numerics followed by three alphabetic characters (seven characters altogether).

Question 8.5. How many characters do the following patterns anticipate in the value of N:

(a) `N?1A`

(b) `N?2L`

(c) `N?1A2L`

(d) `N?2L1A`

(e) `N?1"A"3U`

(f) `N?1"NO"`

The pattern-match operator usually appears in the argument of an IF command. For example:

```
READ !,"DO YOU AGREE (Y OR N)? ",ANS IF ANS?1"Y" DO YES
```

prompts the user to answer a question that needs only a Y for yes or N for no. If the user responds with the letter Y then execution branches to the line labelled YES. Note that in this example the check on ANS could be written more concisely as:

```
IF ANS="Y"
```

The power of the pattern-match operator is extended by the ability to specify that an indefinite (rather than a precise) number of characters may occur. This is done by using the period (.) as in:

```
IF ANS?1"Y".E
```

In this example the value of ANS is checked to see whether it begins with the letter Y (denoted by the pattern 1"Y") and whether it continues with any number of any characters whatsoever (denoted by the pattern .E).

Another example is to check that the value of NUM is one or more numerics using the following code:

```
IF NUM?1N.N
```

where the pattern 1N ensures that at least one numeric is present and .N allows for any number (including zero) of other numerics.

Question 8.6. What values for the variable X are sought with the following instructions:

```
(a)  IF X?1P

(b)  IF X?2N1"-"3L

(c)  IF X?1U.A

(d)  IF X?.N1P.A
```

It is also possible to check for a range by combining the precise number with the period as:

```
IF LIN?1.5N
```

This example checks that the variable LIN contains at least one numeric and no more than 5 numeric. Both the lower limit and the upper limit may be omitted:

```
IF TEXT?.10A

IF TEXT?20.A
```

In the first case, the variable TEXT must contain zero to 10 alphabetic characters (a null string would make the statement true). In the second case, the variable TEXT must have 20 or more alphabetic characters.

Question 8.7. What values for the variable Y match the patterns in the following instructions:

```
(a)  IF Y?1.N

(b)  IF Y?2A1". "1.A

(c)  IF Y?.5N.8A

(d)  IF Y?1U2.10A.E
```

The negation operator (') may precede the pattern-match operator, allowing the programmer to give an explanatory message if the user's answer does not conform to the preferred format. It is usually simpler to check for a wrong answer, give an error message, and branch back to repeat the question, rather than to check for a correct wrong. An example is the following line labeled NENTRY:

```
NENTRY READ "NUMBER? ",NUM IF NUM'?IN.N WRITE "ENTER DIGITS
ONLY",! GOTO NENTRY
```

If the value of NUM does not comprise one or more numerics, a message is displayed and execution branches to line NENTRY where the user is asked to re-enter the value for NUM. The code is much clearer than that created by omitting the negation operator, especially as another GOTO command is then needed:

```
NENTRY READ "NUMBER? ",NUM IF NUM?1N.N GOTO NEXT
       WRITE "PLEASE ENTER DIGITS ONLY",!  GOTO NENTRY
NEXT   ...
```

Question 8.8. Write code to do the following:

(a) Check whether the value of the variable ANS begins with the letter N, in which case branch to the line NUMERIC.

(b) Check that the value of ART is not a punctuation character; if it is not, explain that it should be and branch to the line SHOW.

(c) Request on new lines values for ITEM 1, ITEM 2, and so on; stop if the entered value is not one or more upper-case alphabetics.

Occasionally it may be necessary to check whether a certain pattern contains more types of character than one of the simple groups (such as alphabetics) introduced above. For example, suppose it is necessary for the variable CAN to be either a numeric digit or a punctuation character. Instead of testing for the numeric format:

```
IF CAN?1N DO GOOD
```

and then for the punctuation format:

```
IF CAN?1P DO GOOD
```

MUMPS can combine the two patterns thus:

```
IF CAN?1NP DO GOOD
```

In this case the branch to GOOD is executed whenever CAN consists of one numeric digit or one punctuation character (but not both). The same happens when the letters N and P are reversed:

```
IF CAN?1PN DO GOOD
```

Question 8.9. Write code to check whether the variable ROW comprises an upper-case alphabetic followed by two characters that may be lower-case alphabetics or numeric digits.

8.3. LOGICAL OPERATORS & (AND) AND ! (OR)

A series of truth-valued expressions may be linked by means of the AND (&) operator. If each of the linked expressions is true then the combination is true. However, if at least one of the expressions is false then the combination is false. For example suppose you wanted to check that variables X and Y each had a value of a single digit, suitable code would be:

```
IF X?1N&(Y?1N) WRITE "BOTH X AND Y ARE ONE-DIGIT NUMERICS"
```

This code may be read as saying "if X has the format of one numeric and if Y has the format of one numeric, then write out the message". The message in the quotation marks is displayed only if the values of X and Y are both single-digit numerics. Because MUMPS evaluates expressions on a character-by- character basis from left to right, parentheses are placed around Y?1N to show MUMPS what is the logical intention; some programmers would also place parentheses around X?1N to improve the readability of the code, although in this case the meaning of the code would not be affected by the introduction of the parentheses.

Question 8.10. In the above example, for which of the following would:

```
BOTH X AND Y ARE ONE-DIGIT NUMERICS
```

be displayed?

(a) X=1 and Y=2

(b) X=0 (zero) and Y=9

(c) X=9 and Y=10

(d) X=9 and Y="A"

(e) X=8 and Y="X"

(f) X=3 and Y=X

(g) X=10 and Y=X

Question 8.11. Write code to test that the value of NAME follows alphabetically the letter "M" and that it has the format of one upper-case alphabetic followed by any number of lower-case characters.

Question 8.12. Note the parentheses after the & operator in the example at the start of this section. What would happen if the parentheses were omitted?

Another way to link a series of truth-valued expressions is by means of the OR (!) operator. If any one of the linked expressions is true then the combination is true. All of the expressions have to be false to make the combination false. For example, suitable code to check that either the variable X or the variable Y (or perhaps both) have a value of single digits is:

```
IF X?1N!(Y?1N) WRITE "X OR Y OR PERHAPS BOTH ARE ONE-DIGIT
NUMERICS"
```

Question 8.13. In the above example, for which of the following values would:

X OR Y OR PERHAPS BOTH ARE ONE-DIGIT NUMERICS

be displayed?

(a) X=1 and Y=2

(b) X="ONE" and Y="TWO"

(c) X=1 and Y=99

(d) X="A" and Y=3

(e) X= "A" and Y="B"

Question 8.14. Write code to test that either the value of the variable X exceeds 9 or the value of ANS has the format of one or more numerics.

Question 8.15. Note the parentheses after the OR (!) operator in the earlier example. The parentheses instruct MUMPS to change the sequence in which the expression is evaluated from the customary, simple left-to-right precedence. What would happen if the parentheses were removed to produce:

```
IF X?1N!Y?1N WRITE ...
```

Some other examples of checking results are as follows. If the answer to be entered should contain a string from a small list, the $PIECE function can be used with the FOR command. For example, suppose the answer should contain BIG or LARGE or GREAT or HUGE. Possible code to test for that is:

```
FOR J=1:1:4 IF ANS=$PIECE("BIG*LARGE*GREAT*HUGE","*",J) GOTO RIGHT
```

The FOR command may be avoided (and the checking process made faster by:

```
IF ANS="BIG"!(ANS="LARGE")!(ANS="GREAT")!(ANS="HUGE") GOTO RIGHT
```

Alternatively, although perhaps with less clarity, this code could be used:

```
IF ANS'["*"&("*BIG*LARGE*GREAT*HUGE*"[("*"_ANS_"*") GOTO RIGHT
```

Question 8.16. In the last example, for which of the following values of ANS is the line RIGHT branched to:

(a) "BIG"

(b) "BIG*"

(c) "EAT"

(d) "GREAT*HUGE"

Question 8.17. Write code to check whether the value of W includes the characters "GOOD" but excludes the characters "NO".

CHAPTER 9

ASCII CODE: MORE FUNCTIONS AND OPERATORS

9.0. INTRODUCTION

The ASCII coding scheme is used by MUMPS to represent alphabetic, numeric, and punctuation characters (Section 9.1). This coding scheme is basic to the use of the $ASCII function (Section 9.2) and the $CHAR function (Section 9.3), the "follows" operator (Section 9.4), as well as the $ORDER function (Section 9.5). The significance of the ASCII scheme to the pattern-match operator is noted in Section 9.6. Two more arithmetic operators (division with integer quotient and modulo) conclude the chapter (Section 9.7).

9.1. THE ASCII CODING SCHEME

The ASCII (American Standard Code for Information Interchange) coding scheme has been defined by the ANSI (American National Standards Institute) and published by them in their document number X3.4-1977. MUMPS uses the ASCII code. This code is shown in Table 9.1.

Each ASCII code is a number from 0 to 127. Code number 65 represents the upper-case letter "A", code number 66 represents the upper-case letter "B", and so on to code number 90 representing the upper-case letter "Z". Lower-case alphabetics are represented by code 97 for the lower-case letter "a" to code 122 for the lower-case letter "z". The 10 digits zero "0" to nine "9" are represented by ASCII codes 48 to 57.

A variety of "punctuation" or "special" characters is needed in MUMPS, such as the space that separates an argument from its command and the dollar sign that precedes the name of a function or a special variable. ASCII codes from 32 to 47, from 58 to 64, from 91 to 96, and from 123 to 127 are used for punctuation characters. For example, ASCII code number 32 represents the space " " (named SP in Table 9.1). In total 33 codes are used for punctuation characters.

Question 9.1. What ASCII code represents the dollar sign "$"?

The remaining 33 of the 128 ASCII codes are used for "nonprinting" characters. These are sometimes called "control" characters because they control the action of your terminal or its communication. For example code number 7 causes the bell (signified by BEL in Table 9.1) to be rung on your terminal and code number 13 causes the carriage or cursor to be returned to the start of the line (signified by CR in Table 9.1).

Do not be worried if many of the characters represented by the ASCII code do not appear on the keyboard of your terminal. You may be using a terminal that has only the half of the 128 characters appearing in the two central columns of Table 9.1 and corresponding to ASCII codes 32 to 95. The unavailability of the other 64 characters will not usually be a problem.

CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
0	NUL	32	SP	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Table 9.1 Table of ASCII characters, after ANS X3.4-1977. The code values are those that appear as values of the $ASCII function (Section 9.2) and as arguments of the $CHAR function (Section 9.3).

9.2. $ASCII

The $ASCII function selects a character of an expression and returns the ASCII code of that character. For example:

```
$ASCII("A")
```

is 65 because 65 is (from Table 9.1) the ASCII code corresponding to the upper-case alphabetic "A". Similarly:

```
$ASCII("C")
```

is 67.

Question 9.2. What is the value of:

```
$ASCII("E")
```

For the special case of the empty string (""), MUMPS gives a value for $ASCII("") of -1. The argument or expression in the parentheses of the $ASCII function may be a variable instead of a string in quotes. For example if variable VON has the value "A" then the value of:

```
$ASCII(VON)
```

has the value 65. If VON has the value "C" the result is 67.

Question 9.3. What is the value of:

```
$ASCII(NAT)
```

if variable NAT has the value "F"?

The first argument of the $ASCII function may be followed by a second, whose integer value specifies the position in the value of the first argument of the character to be manipulated. For example:

```
$ASCII("JOAN",3)
```

specifies that the ASCII code of the third character in "JOAN" is to be returned, so that the ASCII value of 65 is produced. Similarly:

```
$ASCII(NAME,LETTER)
```

produces the ASCII code for the character at position LETTER of variable NAME. Therefore if name is "DARC" and LETTER is 4, character number LETTER of NAME is "C" and the resulting value of $ASCII is 67. If the second expression is unspecified, its value is assumed to be 1, and the ASCII code of the initial character in the first expression is returned.

One function may be placed inside another, allowing more flexibility (although making the code a little harder to read). For example suppose that the variable NAME contains a person's first or given name followed by a space and then the person's last or family name. To determine the ASCII code for the initial letter of the family name, you could use:

```
$ASCII($PIECE(NAME," ",2))
```

Note the two adjacent closing parentheses, one for the $PIECE function and the other for the $ASCII.

Question 9.4. If T has the value of any line of text, what are the results of:

```
WRITE $EXTRACT(T,$FIND(LINE,".  "))
```

and:

```
WRITE $ASCII(T,$FIND(LINE,".  "))
```

What happens if T has the value "is produced. Similarly:"?

The reader interested in a more advanced use of $ASCII should consult MUG's "Advanced MUMPS Techniques" (see Price List). In the monograph the use of $ASCII is described for converting one set of values to another by means of an "$ASCII table".

9.3. $CHAR

The $CHAR function converts one or more integers into a string of characters whose ASCII codes are those integers. It performs the converse of the $ASCII function. Thus, as we know from Table 9.1 that the character "A" corresponds to the ASCII code of 65, the result of:

```
WRITE $CHAR(65)
```

is that:

A

is displayed. Alternatively:

```
WRITE $CHAR(65,66,67,68,69)
```

produces:

```
ABCDE
```

So does:

```
FOR J=65:1:69 WRITE $CHAR(J)
```

If a variable is used as an expression for $CHAR, its integer value is calculated by the MUMPS system and the corresponding ASCII character is produced. Thus:

```
SET V=$CHAR(N)
```

causes a value to be assigned to the variable V. If N is 65, the assigned value is "A". If N is 66, the assigned value is "B". To assign to V any upper-case alphabetic, N's value may be from 65 (for "A") to 90 (for "Z").

Question 9.5. Suppose M could have the value of 1, 2, 3, or 4, as a result of which you wanted to assign a value to T of "AA", "AB", "AC", or "AD". What MUMPS code would you use to do that?

9.4. FOLLOWS OPERATOR

The follows operator is represented by the symbol]. One value is said to follow another if it occurs later in the ASCII table shown in Table 9.1. Thus the ASCII code serves as a collating sequence for placing expressions in order. In particular, it provides collation of the upper-case letters A to Z followed by the lower-case letter a to z.

For example "OTHELLO" follows "MACBETH" in MUMPS because the ASCII code for O is 79, which is greater than 77 the ASCII code for M. Thus the code:

```
IF "OTHELLO"]"MACBETH" SET A=1
```

assigns to variable A the value 1. Similarly, if variable ^N(1) has the value "ROSCOE", then:

```
IF ^N(1)]"MAC" DO NOTA
```

causes execution of the line labelled NOTA. That is because the ASCII code for the initial character of "ROSCOE" is 82, which exceeds the ASCII code of 77 for "M".

Question 9.6. If Y(1) is "SCAFELL" and Y(2) is "FROSWICK", what is the result of:

```
IF Y(1)]Y(2) SET Y=Y(2),Y(2)=Y(1),Y(1)=Y DO NEXT
```

If the two expressions being compared are the same in the first characters, comparison continues between the subsequent characters of each expression until difference is found or the end of an expression is reached. Thus, when X(1) has the value "MD", it is true that X(1)]"MC". However, if X(1) has the value "MA", it is false that X(1)]"MC". Similarly if N(1) is "ABERAVON", N(2) is "ABERDOVEY", N(3) is "ABERGAVENNY", and N(4) is "ABERDEEN", it is true that:

```
N(2)]N(1)

N(3)]N(1)
```

and

```
N(4)]N(1)
```

Note that it is also true that:

```
N(2)]N(4)
```

although N(2) and N(4) are identical over the first five characters, and therefore false that:

```
N(4)]N(2)
```

It is also false that:

```
N(1)]N(4)
```

or

```
N(2)]N(3)
```

A problem arises when the "follows" operator is used for sequencing numerics. While it is true that:

```
2]1
```

and:

20]19

it is false that:

10]9

because 1 (the initial character of 10) precedes 9 in the ASCII code sequence.

Question 9.7. If variable WORD has the value "THIS", is it true or false that:

WORD]"THE"

How about:

WORD]"THISTLE"

and

WORD]"THAT"

and

WORD]"WORD

9.5. $ORDER AND STRING SUBSCRIPTS

As was mentioned in chapter 5, $ORDER will give the next string subscript in order based on table 9.1. Specifically, the $ORDER will present all numeric subscripts first (in numeric order) and then each string subscript in the ASCII collating sequence. For string subscripts, this can be illustrated by using the following operators:

```
KILL A

SET A("AAC")=1

SET A("AA")=2

SET A("AAF")=3

WRITE $ORDER(A("AAE"))
```

will cause AAF to be written because the subscript AAF follows AAE in the ASCII collating sequence, and it is the only subscript that follows AAE.

```
SET X=$ORDER(A("B"))
```

will give a null value to the variable X because there is no subscript that follows the subscript "B" (remember that $ORDER gives a null value when the last subscript has been past).

Question 9.8. The array TESTING has the following values:

```
TESTING("JANICE") = 96.2                    TESTING("JAMES") = 78

TESTING("typing") = 13                      TESTING("9232")   = 55.3

TESTING("Abby")   = 102.4                   TESTING("17.5%") = 8
```

What would be the results of the following:

a. `SET X=$ORDER(TESTING(""))`

b. `SET Y=$ORDER(TESTING("IBC"))`

c. `SET NUM=$ORDER(TESTING(X))`

d. `SET X=$ORDER(TESING(NUM))`

e. `SET X=$ORDER(TESTING("ZEBRA"))`

f. `SET Y=$ORDER(TESTING(X))`

Question 9.9. Using the same array described in question 9.4, what would be the results of the following code:

```
SET SUB="" FOR COUNT=1:1:6 SET SUB=$ORDER(TESTING(SUB)) WRITE
!,SUB
```

9.6. THE ASCII CODE AND PATTERN MATCHING

The pattern-match operator is represented by the question mark "?", as explained in Chapter 8. It includes a pattern-code capability. This lets you use one or more of the alphabetics allowed by MYMPS to specify different classes of character. The pattern codes, the character classes they check for, and the relevant ASCII codes are not summarized:

Pattern Code	Character class	ASCII codes
A	52 alphabetics (upper and lower case)	65 to 90 and 97 to 122
C	33 control characters	0 to 31; also 127
E	All 128 characters	0 to 127
L	26 lower-case alphabetics	97 to 122
N	10 numerics	48 to 57
P	33 punctuation characters	32 to 47; 58 to 64; 91 to 96; 123 to 126
U	26 upper-case alphabetics	65 to 90

The groupings of the 128 characters into these sets corresponding to the main pattern-match code are shown by the lines on Table 9.1.

9.7. ARITHMETIC OPERATORS

The MUMPS operators for addition, subtraction, and multiplication are +, -, and *, as seen in Section 2.3. For example:

```
WRITE 9+2*2
```

causes:

```
22
```

to be displayed. Remember that MUMPS evaluates expressions from left to right and that parentheses may be introduced to change the order of evaluation, so that:

```
WRITE 9+(2*2)
```

results instead in

```
13
```

A fourth arithmetic operator has also been introduced: the division operator, /. For example:

```
SET N=A/10
```

assigns to the variable N a value that is a tenth the value of the variable A. Similarly

```
WRITE X/4
```

causes a quarter of the value of X to appear so that if X has the value 7 then:

1.75

is displayed.

The fifth arithmetic operator is the "integer-divide" operator "\", which causes division with a resulting integer quotient. This operator occurs between two numbers. It causes the first to be divided by the second and truncates the result to an integer. For example:

```
WRITE X\4
```

results in

```
1
```

whenever X has the value of 4 or more but less than 8;

```
SET N=A\10
```

assigns a value of 1 to N if A is 10, 11, or any value of at least 10 but less than 20; and:

```
WRITE Q\13
```

displays 0 (zero) if Q exceeds -13 but is less than 13.

The final arithmetic operator is modulo, #. Modulo is most simply thought of as the remainder left after a division is performed: the remainder of dividing 4 by 3 is 1 and so the value of 4#3 is 1; the value of 17#5 is 2; and so on. For example:

```
WRITE A#10
```

displays the last digit of A's value. If A has the value of 31, then A#10 is 1. If A is 73, the A#10 is 3. (If negative numbers are used with modulo, the results may not be what you would expect because modulo is not identical to the remainder. Test with care any code that uses modulo.

Standard MUMPS expects numbers to have no more than nine significant digits. Be careful not to exceed this range, particularly when using normal division (with the / operator), if you intend your code to be used on a different MUMPS implementation from the one where you are using it. If you go outside this range of nine digits, the result of a calculation may vary between implementations.

CHAPTER 10

SPECIAL VARIABLES AND MORE FUNCTIONS

10.0. INTRODUCTION

A special variable is distinguished from a local or global variable in that its name of alphabetic characters is preceded by the dollar ("$") symbol. This and other characteristics of special variables are discussed in Section 10.1. The Standard MUMPS special variables are $HOROLOG (Section 10.2), $IO (Section 10.3), $JOB (Section 10.4), $STORAGE (Section 10.5), $TEST (Section 10.6), and $X and $Y (Section 10.7). Three new functions are introduced: $RANDOM (Section 10.8), $SELECT (Section 10.9), and $TEXT (Section 10.10).

10.1. WHAT IS A SPECIAL VARIABLE?

You are already familiar with the local variable, an entity that has a variable name and may be assigned different values. Each local variable is usually available to only one user in a single work space or partition. This contrasts with a global variable, which may be available to several authorized users in diverse work spaces. Note, however, that the value of a local (but not a global) variable may be assigned directly by MUMPS code, using the READ command:

```
READ "VERSION (1 or 2)? ",VER
```

and both local and global variables may be assigned values using the SET command:

```
SET A=1,^CAR(3,B)="ARGON"
```

Remember also the range of characters that may be used for the name of a local or global variable: the first character of a name may be any of the 52 upper-case and lower-case characters or it may be the percent symbol "%"; subsequent characters may be any alphabetic or any of the 10 digits. Of course a global name is preceded by the circumflex symbol "^" to show that a global variable rather than a local variable is being used. Any of the allowed values for data may be assigned to such variables.

A special variable also has a name and a value. Assignment of a value to a special variable is performed by the MUMPS implementation itself, instead of by the programmer's MUMPS code. A special variable's value gives information about the MUMPS system, such as the vertical or horizontal position on the display screen or page at which the next character will appear. Although

you cannot assign a value to a special variable, you can access and use the value of each such variable. Also you can alter the values of several of these variables as an indirect result of other activities. For example, by writing some characters you may alter the special variables that designate the position at which the next character is to appear.

The name of a special variable is restricted to the 52 alphabetics preceded by the dollar symbol "$" with lower case letters always translated to their uppercase equivalent. Remember that a function also uses only upper-case alphabetics for its name, preceded by the dollar symbol. However, a function is always followed by two parentheses containing the expression(s) on which the function acts, whereas a special variable is not followed by expressions in parentheses. An example of a special-variable name is $HOROLOG, discussed in the next section. The possible values of each special variable vary for different special variables and each variable usually has a limited range of possible values.

Question 10.1. Which of the following has the syntax of a special variable, a local variable, a global variable, and a function:

(a) $HOROLOG

(b) ^%D(3)

(c) $LENGTH(FAME)

(d) $X

(e) GOAL

(f) ^XTRA(AT,BT,CT)

10.2. $HOROLOG

Each MUMPS implementation should be able to keep track of the date and time. The value of the special variable $HOROLOG is a two-part string that gives you the present date and time to the nearest second. If you inspect $HOROLOG by entering:

```
WRITE $HOROLOG
```

the response from the computer is two numbers separated by a comma, such as:

```
50470,47585
```

The number before the comma is the number of days since 31

December 1840. Accordingly if the number before the comma is 0, the date should be 31 December 1840, if it is 1 the date should be 1 January 1841, if it is 2 the date should be 2 January 1841, and so on. The value of 52337 corresponds to 14 April 1984.

Why is 31 December 1840 chosen as the "zero" day, you might well ask. The answer is largely "for historical reasons". Obviously some date has to be chosen as the zero point for a Standard like MUMPS, and a date accepted by tradition is often as good as or better than another. For information about the MUMPS Development Committee's discussions concerning the choice of the zero date for MUMPS, contact the Chairman of the MUMPS Development Committee. However, you do not need to remember the zero value of 31 December 1840 because your MUMPS implementation should have a "system routine" that you can invoke to translate from $HOROLOG to the present date. For example, if such a routine is named %DATE, the instruction:

```
DO ^%DATE
```

causes:

```
14 APRIL 1984
```

or whatever is today's date to be displayed. Some code that calculates the current year from the value of $HOROLOG is as follows:

```
D    SET D1=$PIECE($HOROLOG,",",1)
     SET D=D1-(D1<21609),X=D#1461,Y=D\1461*4+1841+(X\365)
     IF X=1460 SET Y=Y-1
     WRITE !,"YEAR IS ",Y QUIT
```

Schullman's routine to generate and write out the weekday, day, month, and year is given and discussed on pages 9 to 13 of the August 1978 issue of the MUG quarterly.

$HOROLOG contains a second number after the comma. That gives the time expressed as the number of seconds since midnight. If that number is 0, the time is midnight. If it is 1, only 1 second has elapsed since midnight. If it is 3600, an hour has elapsed (60 minutes per hour multiplied by 60 seconds per minute). And so on. Most MUMPS systems have various routines that you can invoke to perform the conversion from the number given by $HOROLOG to the time that it represents. For example, suppose that the routine named %TIME would cause the time on a 24-hour clock to be displayed. The instruction:

```
DO ^%TIME
```

should cause

17:23

or whatever is the present time to be shown.

Question 10.2. What is the result of the following code:

```
SET T=$PIECE($HOROLOG,",",2) HANG 9 WRITE $PIECE($HOROLOG,",",2)-T
```

Remember to comment on what happens if the date changes between the two uses of $HOROLOG.

10.3. $IO

At any one time, each user has a particular device at which characters can be written out by the WRITE command or from which they can be read in by the READ command. This device is in current use. It is therefore called the current device. The choice of how devices are identified is made by each MUMPS implementor. A device may be specified by a non-numeric such as "TERM1". It is also common to specify or identify devices by small integers. For example, 0 (zero) might be used to identify the device at which you signed-on to MUMPS. In this case, if you instructed:

```
WRITE $IO
```

immediately after you had signed-on, the result would be:

```
0
```

displayed on your terminal.

10.4. $JOB

Each executing MUMPS process has its unique job number, assigned to it by the MUMPS implementation. The assigned number depends upon the implementation, but it must be an integer (that is, a whole number) and it must be positive. For example, the following are possible job numbers:

```
1
9
346
```

but these would not be acceptable:

```
ALPHA
-3
4.8
```

$JOB is used very little by application programmers, but it appears in some utility routines.

10.5. $STORAGE

The value of $STORAGE denotes the number of unused characters remaining in the user's workspace. For portability of MUMPS applications, the MUMPS Development Committee recommends that a maximum of 4000 characters be available for all code and data that a user might want at any one time. The instruction

```
WRITE $STORAGE
```

causes the integer value to be displayed that specifies the remaining space in terms of number of unused characters. For example, the result might be 2327 at one point, fall to 1894 after some lines of code have been added, and then rise to 2451 after all local variables are killed.

$STORAGE is used primarily to test whether sufficient storage space remains to allow the application to proceed as planned. For example:

```
IF $STORAGE<99 DO STORE
```

or:

```
IF $STORAGE<200 DO ^KILL
```

Question 10.3. If the following instructions are given:

```
WRITE !,"KILLING" KILL  SET SPACE=$STORAGE,A=3
```

will the value of SPACE now exceed, equal, or be less than the value of $STORAGE?

10.6. $TEST

MUMPS computes a truth value in several situations, particularly when the argument of an IF command is evaluated or (as noted in Section 6.6) when an time-out occurs on a READ command. The computed truth value is 0 for a false expression and 1 for a true one and its value is stored in a so-called "Test Switch". The truth value may be inspected by:

```
WRITE $TEST
```

which results in 0 or 1 being displayed, or the truth value may be

used in other ways such as by being assigned to a variable:

```
SET ^GOLD(4,942,T)=$TEST
```

For example:

```
IF X>6 WRITE $TEST
```

either causes 1 to be written out (whenever X exceeds 6) or else the value of $TEST is not written because X is not greater than 6 and so the WRITE command is not executed on the same line as the IF command.

Question 10.4. Suppose X has the value of 6 in the above example. What is the value of $TEST after execution of the IF command?

The value of $TEST is inspected by MUMPS when executing an ELSE command (Section 6.3) and also when executing an IF command without an explicit argument. In the latter case, an IF command with an unspecified argument evaluates as true if the value of $TEST is 1 and the code following the IF command on the same line is executed; otherwise the value of $TEST is false and the remaining code on that line is not executed.

10.7. $X AND $Y

The horizontal position at which the next character would appear if it were now printed is determined from $X. $X gives the horizontal carriage position if the current device is a printing terminal or the horizontal cursor position if the current device is a video terminal. By definition in MUMPS, the first horizontal position is column 0 so that $X is 0 at the start of the line. Thus the instruction:

```
WRITE !,$X
```

causes the carriage-return and a line-feed with 0 being displayed at the start of a new line. Alternatively:

```
WRITE ! FOR J=1:1:9 WRITE $X
```

causes the following to appear at the start of a new line:

```
012345678
```

Question 10.5. What are the values of X1 and X2 after the following code:

```
WRITE !,"ART" SET X1=$X WRITE "ISTIC" SET X2=$X
```

Question 10.6. What is the effect of:

```
IF $X>60 WRITE $PIECE(S," ",1),! SET S=$PIECE(S," ",2,99)
```

The vertical position at which the next character would appear on the current device is given by $Y. Thus $Y gives the vertical position on a page or display screen. It is incremented by one for each carriage-return and line-feed, so that:

```
WRITE $Y,!,$Y
```

causes two numbers to be written on separate lines with the second number one greater than the first number. The value of $Y is set to 0 by the form-feed or new-page operator (first noted in Section 7.2) so that:

```
WRITE #,$Y
```

causes 0 to be displayed. To get to line seven, instruct:

```
WRITE # FOR J=1:1:7 WRITE !
```

ALternatively to get to line seven (or stay put if already beyond there) without using the form-feed operator, instruct:

```
FOR J=1:1:7-$Y WRITE !
```

Question 10.7. The value of $X is 8 and that of $Y is 13. What is the result of the following code:

```
SET X=$X,Y=$Y WRITE "COLUMN",X,!,$Y-Y,"..",X-$X
```

10.8. $RANDOM

The seven Standard MUMPS special variables have been introduced. Chapter 10 concludes with three more functions. Remember that while a function has a name like that of a special variable and is prefixed by a dollar symbol, a function is followed by one or more expressions in parentheses.

$RANDOM is followed by an expression whose integer value is taken and then $RANDOM returns an integer value between 0 and one less than that value. For example:

```
WRITE $RANDOM(N)
```

causes a number between 0 and N-1 to be displayed. Each of the possible numbers has equal probability of occurrence. The result of:

```
WRITE $RANDOM(2)
```

should be 0 half the time and 1 the rest. The result of:

```
WRITE $RANDOM(100)
```

should be 0 to 99, each number occurring on one hundredth of the occasions that RANDOM(100) is used. $RANDOM is often used for initializing variables in simulations, games, and other situations where some uncertainty needs to be introduced. For example:

```
FOR J=1:1:4 SET A(J)=$EXTRACT("ROYGBIV",$RANDOM(7)+1)
```

will assign randomly one of the seven alphabetics to the array elements A(1), A(2), A(3), and A(4). This is because $RANDOM(7) produces 0 to 6, to which 1 is added so that the first to seventh characters are extracted.

Question 10.8. What is the result of:

```
WRITE $RANDOM(1),!,$RANDOM(UN)
```

10.9. $SELECT

$SELECT is a powerful MUMPS function that can be used to replace a series of related IF commands, such as those used in the code shown in Figure 2.2. As for the other MUMPS functions, $SELECT is followed by expressions in parentheses. In the case of $SELECT, the expressions are in pairs, with each pair separated from the next by a comma. For each pair, the truth value of the first expression is to be evaluated until a true expression is found. An example of a truth-valued expression is:

```
T<6.5
```

which has a truth value of 1 (for true) if the value of T is less than 6.5 but otherwise has a truth value of 0 (for false).

When a true expression is found, the other expression paired with it is used as the value of the $SELECT function. The truth-valued expression is separated by a colon from the other expression of its pair. For example:

```
WRITE $SELECT(A>0:"POSITIVE",1:"NEGATIVE")
```

Here the first truth-valued expression is:

```
A>0
```

If A is greater than zero, the expression A>0 is true and the associated expression "POSITIVE" is used as the value of $SELECT, so that:

```
POSITIVE
```

is written on the current device. If A is not greater than zero, the first truth-valued expression is false and the second truth-valued expression must be evaluated. The second truth-valued expression is 1, which is true by definition and the associated expression "NEGATIVE" is used as the value of $SELECT so that:

```
NEGATIVE
```

is written on the current device. Note that at least one of the truth-valued expressions must always be true or an error could result if no true value is found.

Question 10.9. In this example $SELECT:

```
WRITE $SELECT(AGE>65:"SENIOR CITIZEN",AGE>40:"MIDDLEAGED,AGE?15:
"YOUNG",1: "CHILD")
```

there are four truth-valued expressions. (a) What are they? (b) What value of $SELECT is used if the third truth-valued expression is true? (c) What happens if AGE has the value 0?

In the example of Figure 2.2, the seven lines RUN+8 to RUN+14 may be compressed by means of $SELECT. The seven lines are:

```
IF T<6.5 SET POINTS=6 GOTO MILES
IF T<8 SET POINTS=5 GOTO MILES
IF T<10 SET POINTS=4 GOTO MILES
IF T<12 SET POINTS=3 GOTO MILES
IF T<14.5 SET POINTS=2 GOTO MILES
IF T<20 SET POINTS=1 GOTO MILES
SET POINTS=0.5
```

They may be replaced by the single line:

```
SET POINTS=$SELECT(T<6.5:6,T<8:5,T<10:4,T<12:3,T<14.5:2,T<20:1,1:0.5)
```

Notice that the first truth-valued expression is:

```
T<6.5
```

If T is less than 6.5 then the second expression of the pair, 6, is assigned as the value of POINTS. Otherwise successive truth-valued expressions are inspected until a true one is found. If T is 20 or greater, all truth-valued expressions involving T are false and the last truth-valued expression is reached, which is:

```
1
```

This has a truth value of true by definition, so that the value of 0.5 is assigned to the variable POINTS. Notice that if the value of T is less than 6.5, the value of POINTS is taken from the first pair; if the value of T is 6.5 or more but less than 8, the value of POINTS is taken as 5 from the second pair of expressions, and so on.

Question 10.10. Using the $SELECT function, write a single line of code that assigns to the node ^SACK(100,1,3) the value of "BLUE" if the variable COL is less than 45, "YELLOW" if COL is 45 or more but less than 60, and otherwise "RED".

10.10. $TEXT

$TEXT returns a specified line of the current routine, which is the routine in the programmer's work space. The desired line is specified in the parentheses following the $TEXT function. For example, if the current routine is the one shown in Figure 2.2, then:

```
WRITE $TEXT(ZEX2)
```

causes the first line of the routine to be displayed and:

```
SET LI=$TEXT(RUN)
```

causes the text of the third line to be assigned to the variable LI. The value of $TEXT begins with the optional line label and continues with the line-start indicator and the line contents.

In addition to specifying the label of a line, as in the above two examples, you may specify an "offset" or additional number in order to access an unlabelled line. Thus:

```
WRITE $TEXT(RUN+3)
```

causes the sixth line (the one three lines below line RUN) to be displayed.

Question 10.11. What code would you write to assign to local variable X the text of a line eight below one labelled AIM?

Instead of specifying a line label with or without an offset as the argument of $TEXT, you can specify an offset from the start of the routine. For example:

```
WRITE $TEXT(+1)
```

causes the first line to be displayed and

```
SET F=$TEXT(+31)
```

assigns the text of the thirty-first line to the local variable F. The $TEXT function is rarely used except in editing code. It may also be used (although the author discourages this) to access explanatory messages stored as lines of text:

```
INPUT     READ ANS IF ANS["?" DO EXPLAIN GOTO INPUT
          ......
EXPLAIN   FOR I=1:1 SET X=$TEXT(EXPLAIN+I) WRITE !,X IF X="" QUIT
```

Finally, the name of the current routine can be obtained by specifying line "0", as:

```
SET ROUTINE=$TEXT(0)
```

CHAPTER 11

WRAP-UP

11.0. INTRODUCTION

This chapter presents all the remaining features of Standard MUMPS except for three that are reserved for Chapter 12. The OPEN, USE, and CLOSE commands facilitate manipulation of devices attached to the computer (Section 11.1). The LOCK command is a software convention to minimize conflicting demands from the users upon resources (Section 11.2). The READ command can limit the number of characters that are entered (Section 11.3), or get the ASCII code of a single character (Section 11.4). The VIEW command and the $VIEW function allow access to data not otherwise available (Section 11.5). Non-Standard commands may be used by a MUMPS implementation provided that each command has an initial character of Z; similarly, a non-Standard function or special variable must begin with $Z (Section 11.6). The action of the IF command (already seen in Chapter 2) is complemented by the capability of the "post-conditional" (Section 11.7).

Names of commands, functions, and special variables may be abbreviated (Section 11.8). Limitations imposed by Standard MUMPS and some suggestions to facilitate the portability of applications are discussed (Section 11.9).

11.1. OPEN, USE, AND CLOSE

When a user signs on to a MUMPS system he acquires a work space that has ownership of a single device. That device is the terminal by which the user signed on. It is the current device and the only one at which characters can be read in or written out until the MUMPS code executed by the user acquires ownership of and control over another device. "Ownership" means that the user can use the device and nobody else can.

Two commands are used. The first is OPEN, which gains ownership of the device specified in its argument. Each MUMPS implementor chooses his own means of referencing or naming devices connected to his "central processing unit". These devices include each terminal (whether it is a video or a printing device), as well as line printers, magnetic tape drives, and so on. The second command is USE, which specifies that the device in its argument is to be the current device, used for reading and writing data until another USE command is executed.

Suppose the implementor chose mnemonics for his device specifiers, with "T1" to "T3" denoting the three tape drives available. MUMPS code to acquire control of the second tape drive could then be:

```
OPEN "T2"
```

If the second tape drive is available, it is assigned to the task that executed the above code. However, if the second tape drive is already assigned to another user, the task pauses at the argument of the OPEN command until the desired device becomes available.

After a device has been opened, it may be made into the current device. Thus:

```
USE "T2" WRITE ^X(31,3)
```

makes T2 into the current device and writes the value of the named global node on to it. The advantage of having both the OPEN and USE commands is that it separates acquisition of device ownership from actual use of a device. A program may OPEN all devices needed for input and output, and maintain ownership until it runs to completion. Thus whenever a device is needed as the current device, the USE command is all that is needed to switch the current device.

The disadvantage of having both the OPEN and USE commands is that a device may be owned by another task that no longer needs it. Each programmer should be considerate in using the CLOSE command to release each device that he no longer needs. For example:

```
CLOSE "T2"
```

would release the second tape drive. That might appear in such a line of code as:

```
OPEN "T2" USE "T2" WRITE "^ART(A,1)=",^ART(A,1) USE "1" CLOSE "T2"
```

Question 11.1. What effects do the following code have:

```
(a)  OPEN "3" USE "3" WRITE !,"THIS IS 3"
(b)  CLOSE "8" USE "8"
(c)  FOR J="1","2","3" CLOSE J
```

The OPEN command uses the same timeout capability as shown for READ in Section 6.6. The timeout is a numeric-valued expression that appears, as for the READ command, in the argument,

following a device specifier and a colon. For example:

```
OPEN "A":TIME
```

tries to OPEN device A. If device A is available during the number of seconds specified by TIME, then it is OPENed. Otherwise it cannot be captured (and is presumably OPENed by some other user of the system's resources). More than one argument could be specified, each with its own timeout. For instance:

```
OPEN "A":TIME1,"B":TIME2,"C"
```

may cause device A to be OPENed if it is available during the first TIME1 seconds and may OPEN device B if it is available during the TIME2 seconds after device A is either OPENed or timed out; for device C, however, the user waits as long as necessary in order to OPEN it. Execution of a timed OPEN affects the value of $TEST: if OPEN is completed before the timeout is exhausted, $TEST is assigned the value of 1; otherwise it is given the value of 0 (zero).

11.2. LOCK

The LOCK command may take as its argument any local or global variable name. This command is used as a software convention to minimize conflicts between users sharing a user area with the same global variables accessible to both. For example, if the first user of a LOCK command on a MUMPS implementation gives any instance of the LOCK command, such as:

```
LOCK AVE
```

this user may proceed. However, if another user tries to execute the same code in his different work space, he is not able to proceed beyond his instance of the LOCK command. As for READ and OPEN, a timeout (see Section 6.6) may be used for LOCK.

Once the first user has completed all the actions that he wishes, he should execute the LOCK command with no argument, to unLOCK everything. Alternatively, an argument could be given, as in:

```
LOCK B2
```

which releases or unlocks variable AVE and locks variable B2.

It is particularly common for a global variable to be the argument of the LOCK command. For example an application that could revise the node ^G(1,1) from more than one place, might include the instruction:

```
LOCK ^G(1,1)
```

immediately before each instance where a new value is to be assigned to the variable ^G(1,1). After a new value is assigned, a new LOCK command should be given at once so that when the LOCK ^G(1,1) is encountered by another user there is no need to pause. The new LOCK command may have any argument other than ^G(1,1); it may even be argumentless.

The argument of the LOCK command does not have to be the name of a global node to be revised in the LOCK command. Alternatively:

```
LOCK A
```

could be used for ^G(1,1) and other arguments for other global nodes. Or, if A denotes the unique identification of an individual whose data are being processed, the above instance of LOCK can have the effect of LOCKing all nodes with data relevant to this individual. Explicit naming of each global node is wise, where suitable, however, in that it makes the code clearer than it would otherwise be.

Specification of more than one argument of a LOCK command is possible but MUMPS requires the argument to be surrounded by parentheses. For example:

```
LOCK (^G(1,1),A,B,%MIX)
```

specifies four variable names that are to be LOCKed. Without the enclosing parentheses:

```
LOCK ^G(1,1),A,B,%MIX
```

results in %MIX LOCKed but the other three variables unlocked.

Question 11.2. How would you LOCK node ^VA(37), perform subroutine R4 to change the value of this node, unLOCK it, and LOCK node ^VA(38)?

As for the READ and OPEN commands, a timeout may be appended to an argument of the LOCK command. For example:

```
LOCK ^G:30
```

specifies that the global ^G is to be LOCKed if it is available during 30 seconds. Furthermore, the value of $TEST is affected, being set to 1 if the argument of the LOCK command is successful and otherwise to 0. Note that the value of $TEST may be

inspected:

```
LOCK ^G:30 IF $TEST=0 WRITE "SORRY.  ^G IS NOT LOCKED."
```

or the IF and/or ELSE commands without arguments may be used to check whether the LOCK was successful:

```
LOCK ^G:30 ELSE  WRITE "SORRY.  ^G IS NOT LOCKED."
```

Note the two spaces after the ELSE command.

11.3. READ - Fixed Length

As we saw in chapter 1, the READ command is used to enter a value into a variable as:

```
READ X
```

and then you type in the value to be put into the variable X, followed by pressing the RETURN or ENTER key. The READ command can also be used in a slightly different way by having the READ end after a specific number of characters have been entered.

```
READ Y#5
```

will take what ever you type followed by the ENTER, or the first five characters you type, and enter them as the value for the variable Y. This is called a "read count", and means that the READ will be ended by either an ENTER key (the way a normal read is ended) or entering the number of characters specified. In the example above, no more than 5 characters could be entered into the variable Y. Some computers may end the READ after the fifth character is typed, other computers may end the READ when a sixth character is typed (only the first 5 would be kept).

11.4. READ - single character

Another form of the READ command is called a single character read, and is ended when ever any key is pressed. The variable of the READ is preceded by an asterisk ("*"):

```
READ *CHAR
```

will be ended as soon as any key is pressed, however the variable CHAR will not contain the letter (or number or symbol) of the key pressed, but the ASCII code of that character (see Table 9.1).

11.5. VIEW AND $VIEW

The VIEW command is an implementation-specific command. It is made available to programmers by an implementor in order to allow the output of data not otherwise available in Standard MUMPS. Routines being designed for portability between implementations should not contain the VIEW command because it takes a non-Standard argument. The VIEW command is usually not required except in certain systems utilities that should be written by experienced MUMPS programmers.

The $VIEW function, like the VIEW command, is implementation-specific. A MUMPS implementor may wish to allow access to data not otherwise available. This access is possible by the implementor's choice of allowable syntax for the argument in the parentheses of this function. Routines intended to be transferred between MUMPS implementations should not contain the $VIEW function because its possible arguments are not specified by the Standard. The few utilities that need $VIEW should be written by experienced MUMPS programmers.

For the moment, while you should be aware of VIEW and $VIEW, do not try to use them without advice from your installation manager.

11.6. NON-STANDARD COMMANDS, FUNCTIONS, AND SPECIAL VARIABLES

A further capability for extending MUMPS is provided to the MUMPS implementor. He may specify any additional command, function, or special variable that he wishes, provided that he prefaces its name by the letter Z. Thus the implementor might define:

```
ZALPHA
```

as a command that would alphabetize the elements of an array or:

```
$ZBOUND(...)
```

as a function that could return the upper and lower subscript limits of defined nodes in a local or global array. An example of a special variable might be:

```
$ZERROR
```

which could be used to identify what is the most recent error in the code that the user tried to execute, together with the routine name and the line in which that error occurred.

You may find that some of the non-Standard features in your MUMPS implementation are very convenient, and you may wish to use

them heavily. Before you do so, please consider whether you hope that your code will be transferred to other users on other MUMPS implementations. If present efficiencies outweigh concerns about future transfers, non-Standard capabilities may be used. However, if you are concerned about maintaining consistency with Standard MUMPS, minimize your use of commands, functions, and special variables whose names begin with the letter Z. You might also beware of other MUMPS capabilities not discussed in this document but appearing in your implementation.

11.7. POST-CONDITION

The IF command was introduced in Section 2.2. It is used to place a condition upon whether or not the code following the IF command on the remainder of the line is executed. Alternatively, a condition may be placed upon an individual command instead of a whole line. This is done by following a command by a colon and then a truth-valued expression. An example is:

```
SET:^FLAG(3)>1 A=4 DO END
```

The SET command is followed by a colon and then the condition (called a "post-condition"):

```
^FLAG(3)>1
```

If the value of ^FLAG(3) is greater than 1, the value of 4 is assigned to the local variable A. If the value of ^FLAG(3) is 1 or less, then the SET command is not executed. Whatever the value of ^FLAG(3), the DO command is executed. Thus, the truth-valued expression places a condition upon the SET command but not on any subsequent command. The SET command is said to be post-conditionalized because the condition follows the command, instead of preceding it as it does when the condition is the argument of an IF command.

The post-condition may be used on any command in Standard MUMPS except the IF, ELSE, and FOR commands. An example of a post-conditionalized DO command is:

```
DO:A>3 LOOP SET A=1,B=1 READ "ATTITUDE? ",ATT
```

The SET and READ commands are always executed but the DO command may not be. When all commands on a line are to be affected by the conditional expression, the post-condition is replaced by an IF command followed by a truth-valued expression. The above line could then change to:

```
IF A>3 DO LOOP SET A=1,B=1 READ "ATTITUDE? ",ATT
```

In this new line, all three commands (DO, SET, and READ) are

executed if the value of A exceeds 3. Otherwise none of them are executed.

For three commands, a post-condition may be placed after an argument of the command. The commands for which post-conditionalized arguments are allowed are DO, GOTO, and XECUTE. The XECUTE command will be discussed in Chapter 12. An example of a post-conditionalized argument of the DO command is:

```
DO AGE:FLAGA=1 SET SUM=SUM+TAX
```

Here the subroutine starting with the line labelled AGE is executed only if FLAGA has the value 1; but no matter what the value of FLAGA, the value of SUM is always incremented by the value of TAX. Multiple post-conditionalized arguments may be used, as in:

```
DO AGE:FLAGA=1,DEST:FLAGB=2,MODE SET SUM=SUM+TAX
```

The subroutine starting at line MODE is always executed and the value of SUM is always incremented by TAX. However subroutine AGE is executed only if the value of FLAGA is 1 and subroutine DEST is executed only if the value of FLAGB is 2. Another example is:

```
SET:'$DATA(A) A=1 DO JUMP:B<1,FLY:B>1,FLOAT
```

The SET command is post-conditionalized by the $DATA function. The first argument of the DO command is JUMP, and the line labelled JUMP is executed when B is less than 1. The line labelled FLY is executed if B exceeds 1. The line FLOAT is executed regardless. Some programmers feel that post-conditions provide a cohesion to code, while others discourage their use on the grounds that they make the code more opaque. As with all aspects of programming, you should use post-conditions with care to produce comprehensible code.

Question 11.3. Write code that displays HAPPY BIRTHDAY if the value of variable DOB is "TODAY" and will in any case display I AM GLAD TO BE HERE. Answer this question in two ways if possible.

Question 11.4. Write code to branch to a line labelled FOX if AN has the value "RED" and to line ELE if AN has the value "GREY". Otherwise do the subroutine labelled SUB and initialize the variable CATCH to zero.

The $TEST special variable was introduced in Section 10.6. It allows inspection of the truth value most recently computed if that evaluation was made for the argument of an IF command. For example, following execution of these two lines of code:

```
IF A>18 WRITE "ADULT"
WRITE $TEST
```

you either see:

```
ADULT1
```

with the last character being the value of 1 for true that is returned by $TEST, or you see:

```
0
```

with the value of $TEST being 0 for false if A does not exceed 18. In the latter case "ADULT" is not displayed.

Alternatively, suppose that the IF command is replaced by a post-condition on the WRITE command:

```
WRITE:A>18 "ADULT"
```

This has the same effect upon whether "ADULT" is displayed as:

```
IF A>18 WRITE "ADULT"
```

There is a difference with $TEST, however. Its value is unchanged when a post-condition is used on a command or an argument. Thus if the value of $TEST is 0 and A is greater than 18, execution of:

```
WRITE:A>18 "ADULT"
WRITE $TEST
```

causes:

```
ADULT0
```

to appear. Here the last character is the value of 0 for false that is returned by $TEST.

Question 11.5. Which of the following lines of code can change the value of $TEST:

(a) `IF MAX>100 WRITE "ONE HUNDRED HAS BEEN EXCEEDED" GOTO MORE`

(b) `WRITE:ANSWER'?1"Y".E "YOU MUST BE JOKING" GOTO P`

(c) WRITE $TEST

(d) READ "WHAT? ",W GOTO A:W="A",B:W?1N.N,C:W]"B",D

(e) IF B=2 DO:J>4 GET

11.8. ABBREVIATIONS

Names of commands, functions, and special variables may be abbreviated by omitting all upper-case alphabetic characters but the first. In the case of non-Standard names that begin with the letter Z, abbreviation is made to the letter Z followed by however many letters the implementor requires. For example:

SET E=4

becomes

S E=4

Functions and special variables retain their initial $ symbol before their first alphabetic, and so the $LENGTH function becomes $L and the $HOROLOG special variable becomes $H. Nonetheless, names of commands, functions, and special variables are spelled out in the remainder of this book for clarity. Most experienced MUMPS programmers abbreviate their commands and so on. Many MUMPS programming groups require their programmers to use these abbreviations. On the other hand, some people argue that the time saved in both the writing and the executing of code thus abbreviated is unimportant compared with the time lost by a programmer trying to revise such compressed code. You should use whichever you prefer or whatever is preferred by the group (if any) where you are learning.

Question 11.6. Convert the first five lines of the code in Figure 2.2 into an abbreviated form:

Question 11.7. Expand the commands, functions, and special variables in the following code:

```
R !,"GOOD MORNING.  HOW ARE YOU? ",ANS,! I ANS="" Q
I ANS["OK" W "I AM GLAD" G Y
I ANS["BAD"!(ANS["POOR") W "I AM SORRY YOU FEEL BAD" G N
I ANS'?1A.A W "ALPHABETIC ANSWER PLEASE" G RETRY
S T=$H D ^QUOTA W !,"TIME = ",$P($H,",",2)-$P(T,",",2) Q
```

11.9. PORTABILITY REQUIREMENTS

Some limitations have been placed upon various aspects of Standard MUMPS in order to encourage the transferability of MUMPS applications. A string is limited to 255 characters. A line of stored code is also limited to 255 characters, with the label of the line and a "line-start" indicator being included in this space, together with any code or comments. The line-start indicator is to be one space for applications when being transferred.

Variable names and routine names may begin with any of the 26 upper-case alphabetics or with the % symbol. Subsequent characters may be any upper-case alphabetics or any numerics. Standard MUMPS uses only the first eight characters of variable and routine names to identify and distinguish names containing nine or more characters. For simplicity, you should restrict variable and routine names to a maximum of eight characters. For global variables, the circumflex symbol (^) that denotes a global is not counted as one of the eight characters. For any subscripted variables, the parentheses and subscripts contained are of course not included in the variable name.

A line label may begin with an upper-case alphabetic or with the percent symbol (%). The subsequent characters may be any of the 26 upper-case alphabetics or any of the 10 numerics. A line label may also be composed entirely of numerics. Standard MUMPS identifies and distinguishes labels containing nine or more characters on the basis of their first eight characters. For simplicity, you should restrict your line labels to a maximum of eight characters.

A number should have no more than nine significant digits. If it has more than nine significant digits, then the application containing it risks being not transferable. The absolute value of each number should lie between 10^{25} and 10^{-25}, -10^{25} and -10^{-25}, or else be zero.

A local of global variable may have any number of subscripts, however, the total length of the variable name plus the length of each subscript plus each comma must not exceed 63. Any individual subscript must not be more than 31 characters long. Subscripts can be made up of any printing characters (including the space character), but not the control characters (see Table 9.1). The null character ("") may be used as a subscript only when "priming" the $ORDER function.

A string literal is an expression atom that is designated by a quotation mark (") before its first character and after its last

one. Between the quotation marks there may be nothing, or a pair of quotation marks, or any of the remaining 94 printing characters of the ASCII code. Those characters are the 52 upper-case and lower-case alphabetics, the 10 numerics, and the 32 punctuation characters (excluding the quotation mark from the total of 33 punctuation characters). Possible string literals are:

```
"BOTHER"      with value BOTHER
""BOTHER""    with value "BOTHER"
"3"           with value 3
"A9"          with value A9
"^X(3,2)"     with value ^X(3,2)
```

Note that in the second example, the outer pair of quotation marks surrounds the string ""BOTHER"". Each double quotes signifies one quote appearing inside the string.

A numeric literal is an expression atom also, but its first character must be either a digit, or a decimal point, or a plus or a minus sign. The mantissa or first part of the numeric literal is a sequence of digits optionally followed by a decimal point and a sequence of digits; alternatively the mantissa may be a decimal point and a sequence of digits. The mantissa is followed by an exponent, which begins with an upper-case letter E, an optional plus (+) or minus (-) sign, and a sequence of digits. An example of a numeric literal is 4.37E5, which denotes 437000.

A MUMPS routine, including the code itself, all local variables, and temporary results, should not require more than 4000 characters, to ensure portability to other MUMPS implementations.

CHAPTER 12

THE XECUTE COMMAND, THE INDIRECTION OPERATOR, AND THE JOB COMMAND

12.0. INTRODUCTION

One of the most controversial capabilities of the MUMPS language is the execution of data as code by means of the XECUTE command (Section 12.1) and the "indirection" operator (Section 12.2). These features are powerful but add tremendous complexity to MUMPS code for the beginning programmer. Accordingly while these capabilities must be presented so that you understand the entire MUMPS language, their use should be quite restricted. The JOB command is the last command presented (Section 12.3). It is used to start a routine processing in a work space other than the current one.

12.1. XECUTE

The XECUTE command may take as an argument any expression, such as the name of a local, global, or special variable, a function, or a string or numeric literal. An example of the XECUTE command is:

```
XECUTE A
```

The value of the variable A is treated as a line of code that is executed as if A's value were inserted into the routine in the place of the XECUTE command and its argument. Thus if A has the value:

```
"WELL DONE!"
```

the instruction:

```
XECUTE A
```

causes:

```
WELL DONE!
```

to be displayed. Alternatively if A has the value:

```
"SET B=4,C=3*X"
```

then:

```
XECUTE A
```

causes the values of B and C to be set equal to 4 and to 3 times the value of X respectively.

Question 12.1. What is the result of:

```
XECUTE NOTHING
```

if the variable named NOTHING has the value:

(a) `SET NAME="SMITH"`

(b) `WRITE !,"THIS WAS EXECUTED"`

(c) `DO ^END WRITE "ANSWER = ",ANS`

In the above examples, the argument of each example of the XECUTE command was a local variable. Alternatively it could have been a global variable as in:

```
XECUTE ^MARJ("CODE",8)
```

or:

```
XECUTE ^MARJ("CODE",B)
```

The latter example provides for additional flexibility because the argument of the XECUTE command depends upon the value of the local variable B. If ^MARJ("CODE",1) has the value:

```
READ !,"NAME ",NAME
```

and ^MARJ("CODE",2) has the value:

```
READ !,"AGE ",AGE
```

then if B has a value of 1, the XECUTE command causes NAME to be requested whereas if B's value is 2 then AGE is asked for.

Furthermore, a function may be an argument of the XECUTE command. For example, if STRING has the value:

```
WRITE A*READ !,"B=?",B*KILL*SET SUM=SUM+X
```

then:

```
XECUTE $PIECE(STRING,"*",I)
```

has an effect that depends on the value of the variable I. When I is 1, the value of A is written. When I is 2, the value of B is prompted for. When I is 3, all local variables are KILLed. When I is 4, the value of SUM is incremented by the value of X. If I has any other value the argument of the XECUTE command will be

empty string, which results in nothing being executed.

Question 12.2. How would you assign to the variable STRING the value used above?

Question 12.3. If S has the value:

```
IF J>4 GOTO L
```

what is the result of:

```
XECUTE S GOTO MORE
```

when (a) J's value is 21 and (b) J's value is 4.

The main value of the XECUTE command is its ability to modify, in effect, code as it is being executed. The resulting flexibility is usually achieved through the use of one or more local or global variables, as in the above examples. Among the problems of using the XECUTE command are the difficulties of (a) assigning the correct values, particularly in terms of getting the syntax correct, to the variables you plan to use and (b) ensuring that the values you need to use are not changed between the time you assign them and the time you need to use them. Another problem is the need for appropriate documentation of such variables and their uses for the XECUTE command. The operations performed by code thus executed indirectly cannot usually be discerned from simply reading the code. This makes debugging and maintaining such code very difficult.

12.2. INDIRECTION OPERATOR

The indirection operator is the @ symbol. It is usually followed by an expression atom, which is a simple expression such as a variable name. The @ operator plus its argument comprises an instance of indirection, and this may appear in any argument except that of the FOR command.

There are four types of indirection using the @: "argument", "name", "pattern", and "subscript". In argument indirection, the indirection takes the place of a command argument and the result of the indirection must be one or more complete arguments, as in:

```
SET @PIN
```

The result depends on the value of PIN. If PIN has the value:

```
A=3
```

then execution of the above SET command causes the value of 3 to be assigned to the variable A.

Remember that each command may take more than one argument, as in:

```
WRITE EXP," TRUE (1) OR FALSE (0) -- ",@EXP
```

Here EXP is some truth-valued expression such as:

```
%X=3
```

Its truth or falsehood is specified as a value of 1 or 0 resulting from execution of @EXP. The value of EXP is a comparison of the value of %X with the integer 3. If %X has the value 3, then the equality is true and the value of @EXP is 1. Otherwise the value of @EXP is 0 (zero). Thus, if %X has the value 3.0001, the above WRITE command results in:

```
%X=3 TRUE (1) or FALSE (0) -- 0
```

Another example of "argument indirection" is:

```
DO @$CHAR(N)
```

which causes a subroutine to be executed starting at the line whose label is the value of $CHAR(N). Thus if N is 65, the first line of the executed subroutine is labelled A; if N is 67, the first line is labelled C; and if N is 37, the first line is labelled %.

Question 12.4. What are the results of:

(a) WRITE A when A has the value "2+3".

(b) WRITE @A when A has the value "2+3".

(c) GOTO LINE when LINE has the value "FAR".

(d) GOTO @LINE when LINE has the value "NEAR".

Alternatively, the @ symbol may occur in any place where a named variable can occur. In this case the value of what follows the @ must be a complete variable name. This is called "name indirection". An example is:

```
WRITE $ORDER(@GLOBAL)
```

Syntactically GLOBAL must be the name of a local or global variable. If GLOBAL has the value ^MARK(A,B) and the node ^MARK(A,B+1) is defined, then the result of the WRITE command is the value of B+1. If A or B is undefined an error should be indicated by the MUMPS implementation.

Question 12.5. The variable STRICT has the value "CLASS" and the variable B has the value "S". Also CLASS has the value "VERY STRICT" and S has the value "STRICT". What are the values of:

(a) $FIND("STRICT","B")

(b) $FIND("STRICT",B)

(c) $FIND("STRICT",@B)

(d) $FIND(STRICT,"B")

(e) $FIND(STRICT,B)

(f) $FIND(STRICT,@B)

(g) $FIND(@STRICT,@B)

Thirdly the @ symbol may introduce indirection in the place where a pattern is required following the pattern-match operator. For example:

```
IF NAME?@X DO FOUND
```

causes execution of the line FOUND if the value of NAME has the pattern specified by the value of X. Thus if X is:

```
1A.A
```

then NAME is checked to see if it is made up of one or more alphabetics. If X is:

```
2N
```

then NAME is inspected to see if it is two numeric digits.

Question 12.6. You want to use a subroutine to perform some calculations and also check the format of a variable named CORAL. The pattern for CORAL is given by X(1) when the variable FLAG is defined and X(2) otherwise. Write the part of the subroutine that shows pattern checking using pattern indirection.

Question 12.7. Write an alternate answer to Question 12.6 that avoids using pattern indirection. You may have to comment on changes to code outside the subroutine as well as inside it. Consider which alternative leads to more understandable code.

The fourth type of indirection is used as an extension to the name indirection when refering to variables. Using the previous example:

```
WRITE $ORDER(@GLOBAL@(C,D))
```

and GLOBAL has the value ^MARK(A,B) and the node ^MARK(A,B,C,D+1) is defined, then the result of the Write command is the value of D+1. As with the name indirection, if any of the variables A, B, C, or D are undefined, an error should be indicated by the MUMPS implemenatation.

Question 12.8. What are the results of the following where FIND has the value of ST(1,C,.3):

(a) SET C="T",@FIND@(7)="First"

(b) SET @FIND@(C)="Second"

(c) SET C=2.34,LONG=19,@FIND@(22,LONG,24)="middle"

(d) KILL LONG

(e) KILL @FIND@(22)

(f) KILL C WRITE @FIND@(7)

12.3 JOB

Sometimes it is desirable to be running more than one program at a time. This may happen, for example, when large file is to be looked through for certain data, and at the same time you want to continue working on another program. With the JOB command it is possible to start another MUMPS process running and still continue working with the current program. The argument for the JOB command is the name of a routine to be run. As an example:

```
JOB ^SEARCH WRITE "SEARCH HAS BEEN STARTED"
```

Will start the routine named SEARCH running in a different work space, and then go on to write out the message "SEARCH HAS BEEN STARTED" on your terminal. The routine SEARCH, meanwhile, will begin working at the first line of the routine, and continue

working until it has completed. There is no further communication between your work space and the work space that the SEARCH is running in.

The argument for the JOB command can be any argument that could be used in the GOTO or DO commands. Thus, a new process could be started at some point other than the beginning of the routine by adding a label to the front of the routine name, as:

```
JOB START^DIP
```

In this case, the routine DIP would be started in another work space, but the processing would begin at the beginning of the line labeled START.

When you use the JOB command to start a new process, it is important to realize that the new process will not have access to any of the variables that exist with your current process. You should be careful when writing a routine that you will later start with the JOB command, to make certain that it can get all the information it needs from existing global variables. Also, if the started routine is to use any devices (such as printing to a terminal), remember that the routine must OPEN and USE the device.

Question 12.9. Write the code that would start a new process in another work area, and begin working at the label ENTER of the routine NOW.

The Job command uses the same timeout capability as shown for the READ in Section 6.6. The timout is a numeric-valued expression that appears, as for the READ command, in the argument, following a routine name and a colon. For example:

```
JOB ^XUQMU:5
```

tries to start the routine XUQMU. If the routine can be started within the number of seconds specified by TIME, then the process has been successful and the value of $TEST is 1. Otherwise, the new task has not been started and the value of $TEST is 0 (zero).

APPENDIX 1

MUMPS PROGRAMMING MANUALS

A variety of documents on MUMPS programming have been produced by the MUMPS Users' Group and the MUMPS Development Committee. These are intended for the MUMPS programmers of varying experience, ranging from the novice to the skillful. In 1984, the major documents are as follows:

1. ANSI MUMPS, by the MUMPS Development Committee. This is the complete technical description of Standard MUMPS as accepted by the MUMPS Development Committee and the American National Standards Institute. An earlier version, edited by J.T. O'Neill, was published by the American National Bureau of Standards. The ANSI MUMPS document gives the precise specification of MUMPS. It is intended more as a guide to implementors of the MUMPS language and operating system than as a teaching manual for application programmers.

2. Standard MUMPS Pocket Guide, by Joel Achtenberg, revised by George Timson. This small booklet is a concise summary of the capabilities of MUMPS. It is intended as a quick reference for a MUMPS programmer who wants to be reminded of how any component of the language is used. Again, it is not intended primarily as a teaching document.

3. MUMPS Programmers' Reference Manual, by Melvin E. Conway. The Reference Manual is a large "dictionary" that describes and illustrates in detail the diverse uses of all MUMPS features. As with the Pocket Guide, the Reference Manual is intended for use by programmers with some experience. It provides them with the more detailed information about a known feature.

4. MUMPS Primer, by R.F. Walters, J. Bowie, and J.C. Wilcox. The Primer is a complete discussion of Standard MUMPS. It is intended to take a user, knowing nothing about MUMPS, into a condition in which he can write MUMPS code and even design applications.

For the most recent titles and prices, contact the MUG office.

APPENDIX 2

IMPLEMENTATION-SPECIFIC FEATURES

You will find it helpful to complete figures A2.1 and A2.2 for your MUMPS implementation. On the rest of this page, list any features of MUMPS that are extensions to Standard MUMPS and what you put in those figures. If you are in any way uncertain about implementation-specific features, be sure to check with your system manager.

1a. What keys do you press to tell the computer that you want to start programming?

1b. How should the computer replay?

2a. What is the name of your user area?

2b. How is the entry key marked on your terminal? (You do remember to press it each time you have finished, don't you?)

2c. How does the computer acknowledge your statement of the user-area name?

3a. What is your password? (NOTE: you probably should not write down the password, but you may. In any case, do remember it.) Also put here all other interactions that must occur before you can write MUMPS code.

3b. What symbol does the computer reply with, to show you that you may now write MUMPS code?

1. How do you designate that code is to be stored for later use rather than executed immediately (see Section 2.4)?

2. How do you store for subsequent use a routine named EXAMPLE (see Section 3.1)?

3. How do you copy the routine named EXAMPLE from long-term storage into your working space (see Section 3.2.)?

4. How do you print (see Section 3.3):

   a. All lines of routine EXAMPLE in your working space?

   b. Selected lines A, B, and C of the routine in your working space?

   c. Lines from A to A+9 inclusive of the routine in your working space?

5. How do you delete (see Section 3.4):

   a. All lines of routine EXAMPLE in your working space?

   b. Selected lines A, M, and Y of the routine in your working space?

   c. Lines from the third to the sixth inclusive of the routine in your working space?

6. How do you insert a new line after a line labelled LINE (Section 3.5)?

7. How do you invoke and use your code editor (Section 3.6)?

APPENDIX 3

SOLUTION TO PROBLEMS

CHAPTER 1

1.1 A MUMPS command is a word that specifies the type of action the programmer wants to have performed. Some commands are WRITE, READ, SET, and KILL. An argument in MUMPS appears after a command and it specifies what is to be manipulated. This is in contrast with a command, which specifies the manipulation.

1.2 A constant is a number of a string of characters whose value does not change. A variable is an entity to which a value may be assigned and whose value may change.

1.3 WRITE X will cause the value of the variable X to be displayed READ X waits for the user to enter a value, and then assigns that value to variable X.

1.4

a. the characters between the quotation marks are displayed.
b. the computer waits for the user to enter a value, which is then assigned as the value of the variable named X.
c. the characters between the quotation marks are displayed, then the computer waits for the user to enter a value which is then assigned as the value of the variable named X.
d. an error message is displayed. The KILL (without arguments) removes all variables, including the variable named A. The value of the variable A can not be written out because the variable no longer exists.
e. if either the variables A or B do not exist, or the variable B has a value of 0 (zero), an error message is displayed. If both A and B are defined, and B is greater than or less than 0 (zero), then the variable named % is given the value that is the percentage that A is of B. The value of the variable name % is then displayed.

1.5 There are usually several different ways of writing code to perform the same task. If your answer is different than the one given, try your answer on your MUMPS terminal.

a. READ "NAME? ",CALL
b. KILL SET B=2,SPORT="RUNNING"

CHAPTER 2

2.1 IF SHIP="TITANIC"

2.2 False. SUM has the value of 97; 97 is larger than 40; therefore SUM<40 is false.

2.3 The READ command is executed only if the variable NAME has the value "AWFUL" (if the variable NAME has any other value, the READ command is not executed).

2.4 HEIGHT'>60 The whole IF statement would be: IF HEIGHT'>60

2.5
a. GOTO ^SUNSHINE
b. GOTO 1^SUNSHINE
c. GOTO EGGS
d. GOTO %^CATASTRO

CHAPTER 3

3.1 ZPRINT MILES,SET+2

CHAPTER 4

4.1 The value assigned to LA3 is 7 when ART has the value "STUDENT" and it is 3 if ART has the value "2*3".

4.2 READ "AGE? ",AGE IF $LENGTH(AGE)=1 WRITE "YOU ARE DOING WELL FOR SUCH A YOUNG ONE"

4.3 5

4.4 6
7
0 (zero); Note that "GROUP" is entirely upper-case characters and is not found in the value of "MUMPS Users' Group".

4.5 X=6
Y=19
Z=19

4.6 E1=1
E2="." (the single character period)
E3="B" (the single character B)

4.7 AUSTIN,FORD

4.8 the variable M is the expression being analyzed; the delimiter is a single period ("."); the second field is being sought.

```
  I NEED DATA
```

is displayed (including the two spaces at the beginning).

4.9

a. "JOINT"
b. "423"
c. "NT*W"
d. "23"
e. "JOINT*WILSON"

4.10

a. the variable B is given the value of "72"
b. "LON--WINDSOR" is displayed
c. the variable S is given the value of "LONDON/WINDSOR"
d. "STAR --A9" is written out

4.11 "3rd"

CHAPTER 5

5.1 SET ^PALE=JOT

5.2 SET D("BIRTH")="9 OCT 1937"

5.3 KILL ^AGE(21),JAR

5.4

a. first level subscript is 4; no second or third level subscripts
b. first level subscript is 4; second level subscript is "*" (a single asterisk); no third level subscript
c. first level subscript is 4; second level subscript is "*" (a single asterisk); third level subscript is 3

5.5 The variable S will be have the same value as the variable ^TAN("AC-469",3,1); the variable T will have the same value as the variable ^TAN("AC-469",3,2). Then, we write the value of ^TAN("AC-469",3,3), or get an undefined message if it has no value.

5.6 SET ^JAW("TEETH","#")=S__"."__Q__"."__R__"."__MAX

5.7 X(1) = "EGG"
X(2) = 10

5.8
a. A(3) and A(3,7)
b. B(1), B(42,1) and B(42,2)
c. ^C(9) and ^C(9,"STRING",1,4)
d. A(3,7)
e. B(42,1) and B(42,2)
f. ^C(9,"STRING",1,4)
g. ^C(9,"STRING",1,4)
h. none

5.9
a. 11
b. 1
c. 11
d. 0
e. 10

5.10
a. 0
b. 9
c. 9
d. 9
e. 9
f. a null string
g. a null string

5.11
a. 1
b. 1
c. 3
d. 3
e. a null string
f. a null string

5.12
a. 7
b. 3
c. 3
d. 7
e. 8
f. 3
g. 3
h. 2
i. a null string
j. 90

CHAPTER 6

6.1
a. false
b. true
c. true

6.2 for the first example, the variable P is given the value "A" and nothing is written. In the second problem, the only action taken is the READ statement in the second line.

6.3 The ELSE command is associated with the last IF command encountered. If the line SUB had an IF statement in it, the ELSE command would be the results of the IF in line SUB, not the IF directly above the ELSE.

6.4
a. 3, 5, 7, 9, 11, 13
b. 4, 3, 2, 1, 0
c. 1, 2, 8, 9, 10, 11, 12, 13, 14

6.5
a. when variable I = 9, and variable X(9) has been set to "A"
b. when variable J has the value of -8
c. never
d. when variable K has a value greater than 20; the actual value of K will be 22
e. when variable L has a value of 7; note that there are two possible ways of ending the loop - when L equals 7 or L has the last value for the loop (2). In this case, however, L will not get to the values 8 and 2 unless the variable L is changed in the line LOOP.

6.6 READ "AGE? ",YEARS IF YEARS>65 HANG 10 GOTO OLD

6.7 HANG 10 READ "INVENTORY OPTION? ",OPT:60

CHAPTER 7

7.1 ITEM READ !,"ITEM: ",IT IF IT="?" WRITE "ENTER 1 TO 8 FOR THE 8 ALLOWED ITEMS" GOTO ITEM

7.2 NET READ !,"TV NETWORK: ",X IF X=""!'$FIND("ACNP",X) WRITE " A (FOR ABC), C (FOR CBS), N (FOR NBC), OR P (FOR PBS)" GOTO NET

7.3 READ !,"NAME: ",NAME,!,"AGE: ",AGE

7.4 FOR Z=1:1:8 WRITE !,^TEXT(3,J),": " READ ANS(J)

7.5
a. 7
b. 3

7.6 FOR Y=0,2,4,2,0 WRITE !?Y,"*"
Another (less elegant) answer is:
WRITE!,"*",!?2,"*",!?4,"*",!?2,"*",!,"*"

7.7 FOR N=1:1:15 WRITE !,$JUSTIFY(N,2),". " READ X SET ^CARE(4,N)=X

7.8 WRITE !,S,$JUSTIFY(T,10),?40,R

CHAPTER 8

8.1
a. true
b. false
c. true
d. true
e. false
f. false

8.2 IF MEDS1["DIAZEPAM" SET SUM=SUM+1
To check for an exact match, change the "[" to a "="

8.3 IF ANSWER["Y"

8.4 a pattern containing the character A checks for alphabetics. When the pattern contains "A", the upper-case letter "A" itself is checked for.

8.5
a. 1
b. 2
c. 3
d. 3
e. 4
f. 2

8.6
a. 1 punctuation character
b. 2 numeric characters followed by 1 "-" followed by 3 lower-case letters
c. 1 upper-case character followed by 0 or more alphabetic characters

d. 0 or more numeric characters followed by 1 punctuation mark followed by 0 or more alphabetic characters

8.7
a. 1 or more numeric characters
b. 2 alphabetic characters followed by 1 period followed by 2 spaces followed by 1 or more alphabetic characters
c. 0 to 5 numeric characters followed by 0 to 8 alphabetic characters
d. 1 upper-case character followed by 2 to 10 alphabetic characters followed by anything

8.8
a. IF ANS?1"N".E GOTO NENTRY
b. IF ART'?1P WRITE "ENTER ONE PUNCTUATION CHARACTER" GOTO SHOW
c. FOR X=1:1 WRITE !,"ITEM ",X," " READ ITEM(X) IF ITEM(X)'?1.U QUIT

8.9 IF ROW?1U2LN
or
IF ROW?1U2NL

8.10 The line would be displayed in a, b, and f.

8.11 IF NAME]"M"&(NAME?1U.L)

8.12 The code would be: IF X?1N&Y?1N WRITE ...
The X?1N would be evaluated to the truth value of true (1) or false (0). That is combined with Y (the "&Y") to give another truth value or true (1) or false (0). That will always have the format of one numeric. Accordingly, if the parentheses are omitted, the WRITE command is always executed.

8.13 a, c, and d

8.14 IF X>9!(ANS?1.N)

8.15 If the parentheses are omitted the expression is always true and the WRITE command is always executed. That is because the first four characters (X?1N) are evaluated to the truth value of true (1) or false (0); that value is ORed with the value of Y (the "!Y") to give a new truth value that is either 1 for true or 0 for false; that does match with the pattern, specified by the last three characters (?1N).

8.16 The line RIGHT is branched to for value (a) only.

8.17 IF W["GOOD"&(W'["NO")
Alternately, the ampersand may be replaced by a comma and the parentheses omitted: IF W["GOOD",W'["NO"

CHAPTER 9

9.1 36

9.2 69

9.3 70

9.4 The character written will be the first character of the second sentence, assuming that all sentences are separated by a period followed by two spaces. In the specific example, an S will be displayed, and the $ASCII will display an 83.

9.5 SET T="A"_$CHAR(M+64)

9.6 "SCAFELL" follows "FROSWICK" and so it is true that Y(1) follows Y(2). Accordingly the SET command is executed, which switches the values of Y(1) and Y(2) by means of the scratch variable Y, and the line NEXT is branched to.

9.7
```
WORD]"THE"                  true
WORD]"THISTLE"              false
WORD]"THAT"                 true
WORD]"WORD"                 false
```

9.8
a. X = "9232"
b. Y = "JAMES"
c. NUM = "17.5%"
d. X = "Abby"
e. X = "typing"
f. Y = a null string

9.9 The following would be displayed:
```
9232
17.5%
Abby
JAMES
JANICE
typing
```

CHAPTER 10

10.1
a. special variable

b. global variable
c. function
d. special variable
e. local variable
f. global variable

10.2 After a pause of 9 seconds the number 9 is written out because the second piece of the $HOROLOG is the time past midnight. However, if the date changes between the two uses of $HOROLOG the value displayed is -43191, which is 9 minus the number of seconds in a day (9-(12*60*60)).

10.3 The value of $STORAGE is greater before a value is assigned to the local variable A than it is afterwards. The value of SPACE therefore exceeds the second value of $STORAGE.

10.4 0

10.5 X1 = 3 and X2 = 8

10.6 When the carriage or cursor is to the right of column 60, the characters before the first space in the variable S are written out, a carriage-return/line-feed is executed, and the part of S that was written out is removed from the value of S.

10.7 X is assigned the value of 8 and Y is assigned the value 13. The following display appears:

```
        COLUMN8
1..5
```

10.8 0 (zero) is written out, then at the begining of the next line either an integer between 0 and the value of UN-1 appears or an error message is given saying that UN is not defined.

10.9
a. the four truth-value expressions are:
AGE>65
AGE>40
AGE>15
1
b. When the third truth-value expression is true, AGE is greater than 15 but not greater than 40; MIDDLEAGED is written out.
c. If AGE has the value 0 (zero), the last truth-value expression is reached and CHILD is written out.

10.10

```
SET ^SACK(100,1,3)=$SELECT(COL<45:"BLUE",COL<60:"YELLOW",1:"RED")
```

10.11

```
SET X=$TEXT(AIM+8)
```

CHAPTER 11

11.1

a. Control is acquired of device number 3 and the specified code (including the designation of the carriage-return/line-feed formatting operator) is written on to it. If that device is a terminal:
THIS IS 3
is written at the start of a new line. If the device is a storage device, the characters specified by the argument of the WRITE command are stored.

b. An error is indicated. Device number 8 has just been CLOSEd. It cannot be USEd until it has be reOPENed.

c. Devices 1, 2, and 3 are CLOSEd.

11.2 LOCK ^VA(37) DO R4 LOCK ^VA(38)

11.3 WRITE:DOB="TODAY" "HAPPY BIRTHDAY" WRITE "I AM GLAD TO BE HERE"
another way would be:

```
IF DOB="TODAY" WRITE "HAPPY BIRTHDAY"
WRITE "I AM GLAD TO BE HERE"
```

11.4 GOTO FOX:AN="RED",ELE:AN="GREY" DO SUB SET CATCH=0
another way would be:

```
IF AN="RED" GOTO FOX
IF AN="GREY" GOTO ELE
DO SUB SET CATCH=0
```

11.5 a, and e

11.6

```
ZWX2  ;ZIM;28 AUG 79; COOPER'S AEROBIC POINT SYSTEM (1970)
      K  S SUM=0
      R "HOW MANY MILES DID YOU RUN? ",MILES,! I 'MILES G SWIM
      I MILES>100 W "UNLIKELY",! G RUN
      I MILES>2 W " VERY GOOD",!
```

11.7

```
READ !,"GOOD MORNING.  HOW ARE YOU? ",ANS,! IF ANS="" QUIT
IF ANS["OK" WRITE "I AM GLAD" GOTO Y
IF ANS["BAD"!(ANS["POOR") WRITE "I AM SORRY YOU FEEL BAD"
GOTO N IF ANS'?1A.A WRITE "ALPHABETIC ANSWER PLEASE" GOTO
RETRY SET T=$HOROLOG DO ^QUOTA WRITE !,"TIME =
",$PIECE($HOROLOG,",",2)-$PIECE(T,",",2) QUIT
```

CHAPTER 12

12.1
 a. The value of "SMITH" is assigned to the variable NAME.
 b. THIS WAS EXECUTED is written at the start of a new line.
 c. The routine named END is executed, ANSWER = is written, and then the value of ANS appears.

12.2 using the SET command: SET STRING="A*READ !,""B=?"",B*KILL*SET SUM=SUM+X"
using the READ command:
READ STRING to which the user responds with: A*READ !,"B=?",B*KILL*SET SUM=SUM+X

12.3
 a. When J is 21, execution branches to line L. However, after ending the code invoked by the GOTO, MUMPS comes back to the next code after XECUTE S; this is because any argument of an XECUTE command is executed like a subroutine that was DOne. Therefore, execution then branches to line MORE.
 b. When J is 4, the argument (J>4) of the IF command is false so GOTO L is not executed. Instead, execution branches to line MORE. The argument of the XECUTE command is treated as a unit independent from the line in which it occurs, in the sense that failure of the IF command in the argument of XECUTE does not prevent execution of commands following the XECUTE on the same line.

12.4
 a. 2+4 is displayed
 b. 5 is displayed
 c. The line LINE is branched to. (Caught you? This is just the normal GOTO command that you learned before. It is quite possible for there to be a line called LINE as well as a variable called LINE. MUMPS knows which one you mean depending on how you use it.)
 d. The line NEAR is branched to.

12.5
 a. 0
 b. 2
 c. 7
 d. 0
 e. 5
f. 0
g. 12

12.6 SET SUB=2 IF $DATA(FLAG)#10 SET SUB=1
IF CORAL?@X(SUB) ...

12.7 If the values of X(1) and X(2) are known, you could use their values directly in the code. For example, if X(1) is ".A" and X(2) is "3N" the result would be:
IF $DATA(FLAG)#10,CORAL?.A ... (format is correct)
IF $DATA(FLAG)#10,CORAL?3N ... (format is correct)
However, if the value of X(1) and X(2) varies, either you have to use pattern indirection or you must set up an extra line of code to handle each of the additional patterns.

12.8
a. ST(1,"T",.3,7) is given the value of "First"
b. ST(1,"T",.3,"T") is given the value of "Second"
c. ST(1,2.34,.3,22,19,24) is given the value of "middle"
d. the variable LONG is deleted
e. ST(1,2.34,.3,22) and all its descendants are deleted. Assuming that the array ST did not exist before starting this exercise, then the only variable deleted is ST(1,2.34,.3,22,19,24)
f. the variable C is deleted, but the an error message is displayed because the variable C is needed for the WRITE command.

12.9 JOB ENTER^NOW

APPENDIX 4

ALTERNATE CODE

The samples of code shown in this document illustrate various ways of writing in MUMPS. Different programmers and teams develop their own styles, and so you should be aware that there can be many acceptable ways to use MUMPS. For example, many programmers believe that GOTO commands should be avoided if possible. One programmer crystallized such a dislike of the GOTO by suggesting alternate code to that shown in Figure 2.2. His replacement code apears in Figure A3.1. Notice that DO and ELSE commands replace most of the GOTO's in Figure 2.2; also remember how the $SELECT function could be used to specify more briefly how the value of T is to be used to calculate the value of POINTS.

```
ZEX3    ;DDS & ZIM;28 MAR 79;COOPER'S AEROBICS POINT SYSTEM (1970)
START   KILL  SET SUM=0
        READ "DID YOU RUN OR WALK TODAY? ",ANS IF ANS="Y" DO RUN
        READ "DID YOU SWIM TODAY? ",ANS IF ANS="Y" DO SWIM
        READ "DID YOU BICYCLE TODAY? ",ANS IF ANS="Y" DO CYCLE
        HALT
RUN     SET MAX=100,AVERAGE=2 DO DIST IF MILES=0 QUIT
        SET MAX=20,MIN=3.7,AVERAGE=8 DO TIME
        IF (MILES<2)&(T'<20) SET POINTS=0
        ELSE  IF T<6.5 SET POINTS=6
        ELSE  IF T<8 SET POINTS=5
        ELSE  IF T<10 SET POINTS=4
        ELSE  IF T<12 SET POINTS=3
        ELSE  IF T<14.5 SET POINTS=2
        ELSE  IF T<20 SET POINTS=1
        ELSE  SET POINTS=0.5
        SET POINTS=POINTS*MILES
        DO ACCUM
        GOTO RUN
SWIM    ;CODE TO BE WRITTEN
        QUIT
CYCLE   ;CODE TO BE WRITTEN
        QUIT
DIST    READ "NUMBER OF MILES COMPLETED: ",MILES
        IF (MILES<0)!(MILES>MAX) WRITE "UNLIKELY ... TRY AGAIN",! GOTO DIST
        IF MILES>AVERAGE WRITE "   VERY GOOD",!
        QUIT
TIME    READ "HOW MANY MINUTES DID THAT TAKE? ",TIM
        IF TIM'>0 WRITE YOU CAN'T DO THINGS THAT QUICKLY!",! GOTO TIME
        SET T=TIM/MILES IF T<AVERAGE WRITE "   THAT'S FAST",!
        IF T<MIN WRITE "IMPOSSIBLY FAST ... TRY AGAIN",! GOTO TIME
        QUIT
ACCUM   SET SUM=SUM+POINTS
        WRITE "YOU JUST ADDED ",POINTS," POINT" IF POINTS'=1 WRITE "S"
        WRITE " AND YOUR TOTAL IS ",SUM,!,"ENTER MORE",!!
```

Figure A3.1. Alternate code for solution to problem presented in Chapter 2. See also Figure 2.2 on p. 23.

INDEX

Commands	Page Number
BREAK	34
CLOSE	121
DO	23-24, 35, 70-71, 124, 157
ELSE	71-72, 76, 124, 157
FOR	73-75, 124
GOTO	19-21, 70-72, 124, 157
HALT	37
HANG	74
IF	15-19, 71, 76, 123-124
KILL	10-11, 70
LOCK	121-123
OPEN	119-121
QUIT	21-23
READ	8-9, 69
SET	9-10, 69-70
USE	119-121
VIEW	124
WRITE	2, 6, 10, 69
XECUTE	35, 125, 131-133
Z...	124-125

Functions	Page Number
$ASCII	98-99
$CHAR	99-100
$DATA	59-62
$EXTRACT	44-45, 50
$FIND	41-44, 50
$JUSTIFY	82-85
$LENGTH	40-41, 50
$ORDER	62-69
$PIECE	45-49, 102-103
$RANDOM	114
$SELECT	114-115
$TEXT	116-119
$VIEW	124
$Z(...)	124-125

Special Variables	Page Number
$HOROLOG	108-110
$IO	110
$JOB	110-111
$STORAGE	111
$TEST	111-112
$X	112-114
$Y	112-114
$Z...	124-125

Operators		Page Number
Logical	&	91-95
	!	91-95
	'	18-19
Arithmetic	+	104-107
	-	104-107
	*	104-107
	/	104-107
	\	104-107
	#	104-107
	<	17-18
	>	17-18
String	=	17-18
	_	58-59
	[	85-86
	]	85-86
	?	86
Indirection	@	133-136
Formatting	!	79-82
	#	79-82
	?	79-82

Other Terms	Page Number
Abbreviation	128-129
Argument	2, 11
Array	53-59
ASCII code	95-98
Atomic expression	39
Command	2
Constant	7
Expression	39-41
Function	39-52
Global variable	53-54, 107
Implementation specific feature	1, 2-8, 18-19, 29-38, 141-144
Line label	18-19
Local variable	53-54
Naked reference	54-58, 107
Portability	125-129
Post condition	125-128
Special variable	107
String	8-9
Subscript	54-58
Test switch	76
Timeout	75-76
Truth value	16-17
User interaction	77-79
Variable name	7, 53
Variable value	7, 53

COVER DESIGNED BY JACK I. BALLESTERO